Parallel techniques for solving large scale travelling salesperson problems

C P Ravikumar considers four well known heuristics for the travelling salesperson problem and outlines the results of applying parallel randomized search techniques to large examples

As a hard combinatorial optimization problem, the travelling salesperson problem (TSP) has been of pedagogical interest for more than 50 years. More recently, the problem has generated a great deal of practical interest due to its applications in electronic circuit assembly and the drilling of printed circuit boards. In the simplest terms, the TSP is to find a minimum cost Hamiltonian tour of n cities. Since there is no known polynomial time algorithm to solve the TSP, and since n is quite large for practical problems, it is customary to use heuristic techniques and generate suboptimal tours. Even heuristic algorithms are expensive in CPU time when hundreds (or even thousands) of cities are involved. In this paper, we consider four well known heuristics for the TSP and their parallel implementations. Two constructive algorithms are considered: the farthest insertion heuristic and Christofides' approximation algorithm. Two iterative improvement algorithms are considered: the two-opt and three-opt techniques due to Lin and Kernighan. The results of applying parallel randomized search techniques to large instances of the problem are described. We demonstrate the usefulness of parallel processing in solving hard optimization problems by providing experimental evidence for both speedup improvement and an improvement in the quality of the final solutions. The target machines used for these parallel implementations are the Intel iPSC/2 hypercube and the Alliant FX/80.

combinatorial search parallel algorithms Intel iPSC/2 Alliant FX/80 circuit partition travelling salesperson problem

Department of Electrical Engineering, Indian Institute of Technology, Delhi, Hauz Khas, New Delhi, 110016, India
Paper received: 26 January 1992. Revised: 21 April 1992

In this paper, we restrict ourselves to the Euclidean TSP in a plane. The input to the problem consists of an edge-weighted graph on n nodes. Each node represents a city, and an edge (i, j) represents a route from city i to city j. The weight on edge (i, j), denoted c_{ij}, represents the cost of travelling from city i to city j (or vice versa). We are also given the coordinates of the cities in the two-dimensional plane. The cost c_{ij} is the Euclidean distance between cities i and j, namely, $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$. The cost matrix $[c_{ij}]$ is symmetric, since $c_{ij} = c_{ji}$. The problem is to find a minimum-cost Hamiltonian tour of the n cities. A Hamiltonian tour visits each city exactly once. The simplest representation of a Hamiltonian tour is a permutation T of n cities; the tour begins at city T_1 and proceeds to visit $T_2, T_3, \ldots, T_n$, ultimately returning to T_1. The cost of a tour T is given by

$$\text{cost}(T) = \left(\sum_{i=1}^{n-1} c_{T_i T_{i+1}} \right) + c_{T_n T_1} \qquad (1)$$

There are $(n-1)!/2$ possible tours of n cities. For large n, the number of possible tours is exponential in n, making it impractical to look for exact solutions by exhaustive search. Near-optimum tours can be constructed using several available heuristic algorithms such as local search[1], simulated annealing[2] and many others. All these algorithms consume enormous amounts of CPU time when solving large sized TSP instances. For example, a local search technique called 'three-opt' is known to generate tours that are near-optimum, but requires $O(n^3)$ operations per iteration. In an experiment performed by this author, the three-opt technique required three days of CPU-time on a 4 MIPS computer for the 532-city example provided by Padberg and Rinaldi[3]. Practical applications of TSP involve large values of n, as will be

seen shortly. Therefore, it is useful to consider parallel algorithms for TSP. In this paper, we compare the performance of several heuristic algorithms for TSP; these heuristics are briefly explained in the next section. The following section describes parallel search techniques for TSP. Experimental results based on implementation on an Intel iPSC/2 are then reported. Conclusions and directions for further research are provided in the final section.

Applications of TSP

The TSP finds applications in PCB drilling[4] and IC insertion[5]. In high speed mass production of printed circuit boards, the drilling of holes in each board is automatically performed by a numerically controlled tool. The tool must 'visit' each hole position exactly once. Let t_{ij} be the time required to move the tool from the hole position i to position j. If there are n holes in a PCB and these are visited in the order $p(1), p(2), \ldots, p(n)$, then the total time spent in drill movement is given by $T(n) = \sum_{i=1}^{n-1} t_{p(i),p(i+1)}$. In order to minimize $T(n)$, it is essential to solve a TSP in which cities correspond to hole positions and the cost of travelling from city i to city j is the time t_{ij}. The number of holes n can be quite large, up to 3000, since several PCBs are drilled in one step[4].

High speed circuit assembly systems make use of a three-axis placement machine which automatically inserts IC chips into their respective positions on PCBs. Such a machine consists of a chip insertion mechanism and a mechanism to move a PCB into position for insertion. The time to complete the insertion of one chip consists of two chief components: the time to move the board and the time for actual insertion. The insertion mechanism moves vertically during the process of insertion, and the time required for this movement does not vary with chip position. Therefore, to expedite the assembly process, the amount of board movement must be minimized. Once again, this minimization problem can be formulated as a travelling salesperson problem on n cities, where n is the number of chips. The cost c_{ij} corresponds to the movement time of the board from the position of chip i to the position of chip j.

TSP HEURISTICS

TSP heuristics can be classified into constructive and improvement classes. A constructive heuristic builds a tour from scratch, whereas an improvement heuristic accepts a tour as input and improves it iteratively until no further improvements can be made. Examples of constructive heuristics are insertion heuristics such as the farthest insertion algorithm[6], and Christofides' approximation algorithm[7]. Examples of iterative improvement heuristics are the 'two-opt' and 'three-opt' algorithms by Lin and Kernighan[1].

Farthest insertion algorithm

The farthest insertion algorithm begins with a partial tour T consisting of a single city, and adds a new city to T in each iteration. It is clear that the procedure requires $n - 1$ iterations. Iteration i consists of two steps: selection and insertion. The selection step picks an unvisited city which is farthest from the partial tour T. Let v be an unvisited city. Define $d(v)$ as the distance between v and the node in T which is closest to v.

$$d(v) = \min_{u \in T} C_{v,u} \tag{2}$$

Then the selection step chooses the city with the maximum value of $d(v)$ for insertion. If (i, j) is an edge in the partial tour, then inserting a city v between cities i and j will mean deleting edge (i, j) and inserting two new edges (i, v) and (v, j). Therefore the cost of inserting v between i and j is:

$$\delta_{ij} = C_{iv} + C_{vj} - C_{ij} \tag{3}$$

A greedy insertion procedure is to identify cities $t, h \in T$ such that δ_{th} is minimum, and insert v between t and h. To facilitate our discussion of parallel search techniques, we shall distinguish between three types of farthest insertion: deterministic, randomized and parallel randomized. Deterministic farthest insertion (DFI) is one in which the initial city is always chosen in the same way, say city 1. Randomized farthest insertion (RFI) chooses the initial city randomly; each city has probability $1/n$ of being selected as the initial city. Parallel farthest insertion (PFI) makes use of P processors to execute the algorithm, each of which executes RFI starting with a different initial city.

Christofides' approximation algorithm

The chief idea behind Christofides' algorithm for TSP is to first construct a good Eulerean tour of the cities and transform the Eulerean tour through local transformations into a Hamiltonian tour. Let $G = (V, E)$ be the weighted graph representing the travelling salesperson problem. A minimum spanning tree $T = (V, E')$ of the graph G forms a good starting point in constructing a short Eulerean tour. However, T itself does not have an Eulerean tour, since some of the nodes in T will be of odd degree. (Recall that all the nodes of a graph must be of even degree if the graph must permit an Eulerean tour.) Thus, some extra edges must be added to T to achieve the even-degree property; furthermore, the extra edges must add as little to the final tour as possible. Let V_1 be the set of all nodes in T that have *odd* degree; let H be the graph with V_1 as the node set. If we add the edges that correspond to a minimum weight perfect matching of the graph H to the minimum spanning tree T obtained above, then the resulting multigraph will have an Eulerean tour. Let this tour be indicated by $S = v_1 v_2 \ldots v_m v_1$. Then Christofides' algorithm converts S into a Hamiltonian tour using the following procedure.

(1) Scan the tour S from left to right and detect the first node v_f which repeats itself.
(2) If v_f is the last node in the Eulerean tour, then stop; the tour S is Hamiltonian.
(3) Starting from v_f, scan right until a node v_g is found such that v_g does not appear before v_f. Delete all nodes encountered between v_f and v_g. In effect, the tour proceeds directly from v_{f-1} to v_g, rather than taking a circuitous route through v_f.

The time complexity of Christofides' algorithm is bounded by the complexity of the matching step; the latter can be completed in $O(n^3)$ time. The procedure is an approximation algorithm, since it can be shown that

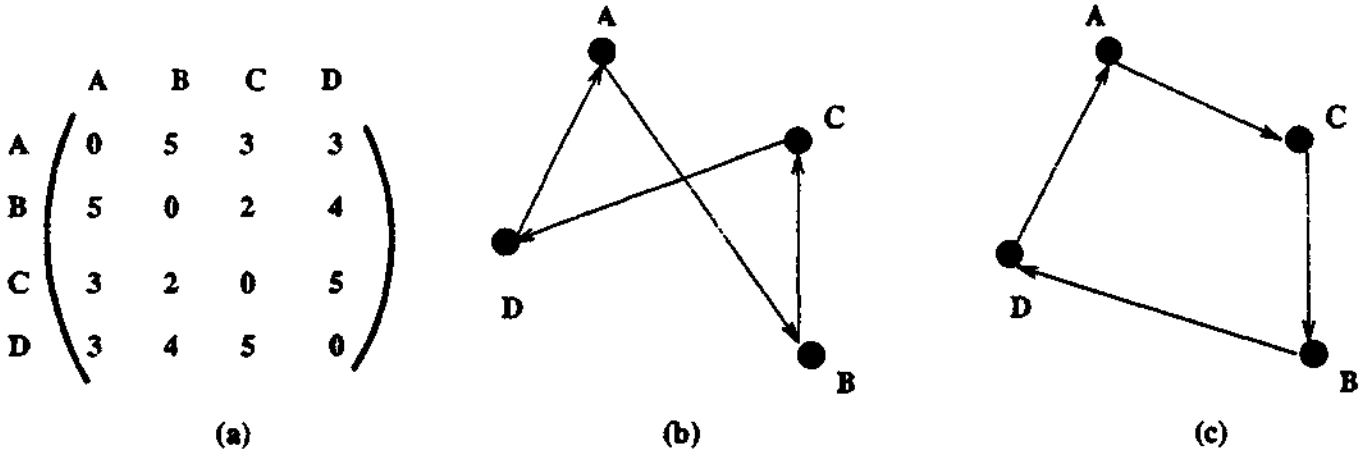

Figure 1. Illustration of the two-opt procedure for a four-city TSP

$$L \leqslant 1.5 \times L^* \quad (1)$$

where L is the length of the tour found by the algorithm and L^* is the length of the optimum tour.

Figure 1(a) shows the cost matrix for a four-city TSP. The cities are indicated by A, B, C, D. The minimum spanning tree T for this example can be constructed using Kruskal's algorithm; the edges in the MST are (BC, CA, AD). The nodes B and D in the MST are of odd degree. A minimum weight matching of these nodes is simply the edge BD. The multigraph obtained by combining the MST and the matching consists of all the four nodes and the edges (BC, CA, AD, BD). An Eulerean tour of this multigraph is BCADB. Since B is the first node which repeats itself and it occurs at the end of the tour, we already have a Hamiltonian tour.

Two-opt procedure

The 'two-opt' procedure is an iterative improvement algorithm for TSP[1]. The procedure takes a Hamiltonian tour T of n cities as input. It may be possible to reduce the length of the tour by swapping the positions of two adjacent cities in T. One iteration of the two-opt procedure consists of examining all possible adjacent pairwise swaps and identifying the pair that gives the highest reduction in tour length. If the highest reduction is negative, the procedure halts; the tour T is a local optimal tour. Otherwise, the tour T is modified by swapping the pair of cities found by the iteration. Figure 1 illustrates the algorithm with a small example of four cities. The cost matrix is shown in Figure 1a. Suppose that the initial tour T is A B C D; the length of the initial tour is 15 (Figure 1b). There are two possible adjacent pairwise swaps: swap B and C, or swap C and D. The former swap results in the tour A C B D with a cost improvement of 3; the latter results in the tour A B D C with a cost improvement of −2. The algorithm selects the former pair and modifies the tour to A C B D; the improved cost function is 12 (Figure 1c). In the next iteration, the possible swap pairs are (C, B) and (D, B); the improvements associated with these pairs are −3 and −5 respectively. Thus the procedure stops and outputs A C B D as the final tour.

In an n-city problem, there are $n(n-3)/2$ possible adjacent pairwise swaps to be examined per iteration. The number of iterations and the cost of the final tour depend on the initial tour. Suppose that the cities are labelled $1, 2, \ldots, n$. A deterministic two-opt procedure (D2OPT) always begins with a specific tour, such as $1, 2, \ldots, n$; it is clear that such a deterministic procedure always terminates in the same local optimum. A randomized two-opt procedure (R2OPT) assigns a random permutation of $1, 2, \ldots, n$ as the initial tour. A parallel two-opt procedure (P2OPT) uses P processors and assigns a different random permutation of $1, 2, \ldots, n$, say T^i, to each processor i; the processors concurrently apply the two-opt procedure. Let F^i be the final tour generated by processor i and let $c(F^i)$ be the cost of F^i. The output of the parallel two-opt procedure is F^i, if $c(F^i)$ corresponds to the minimum of $c(F^i)$, $i = 1, 2, \ldots, n$.

Generalizations of two-opt

The two-opt procedure can be generalized to result in a k-opt procedure. Two-opt uses adjacent pairwise interchange to modify a tour; this amounts to deleting two edges from the tour and adding two fresh sets of edges that result in a different tour. The tour A B C D of Figure 1b is modified to the tour A C B D of Figure 1c by deleting edges AB and CD and adding the edges AC and BD. A k-opt procedure deletes k edges from the tour T and adds a fresh set of k edges so that the resulting graph is still a Hamiltonian tour. It was mentioned earlier that $O(n^2)$ swaps must be examined for every iteration of two-opt. In general, $O(n^k)$ alternatives must be examined in a k-opt algorithm. For $k > 2$, the k-opt algorithm is complicated to implement, since it is non-trivial to maintain the Hamiltonian property of the tour while adding a fresh set of k edges. Even for $k = 3$, the k-opt algorithm is too compute-intensive to be affordable for large n. About three days of CPU time was required to run the three-opt procedure for a 532 city TSP on a single node of an Intel iPSC/2. Nevertheless, the three-opt procedure does result in solutions that are superior to those generated by the two-opt procedure. It is possible to define the deterministic, randomized and parallel versions of a k-opt procedure just as it was done for two-opt. These are denoted DKOPT, RKOPT and PKOPT respectively.

PARALLEL SEARCH TECHNIQUES

Heuristic algorithms for NP-complete optimization problems commonly employ randomization. Informally,

an NP-complete problem is one that a non-deterministic machine (algorithm) is likely to solve in polynomial time. In this sense, a randomized heuristic algorithm can solve optimization problems with a non-zero probability of achieving the global optimum solution. As an example, consider local search heuristics. Typically, a local search algorithm begins with some solution to the optimization problem and improves this solution by making small changes. The initial solution can be generated randomly. One iteration of the algorithm consists of searching for better solutions in the neighbourhood of the current solution: the best of these solutions is then adopted as the current solution. When no better solutions exist in the neighbourhood, the algorithm terminates and reports the current solution as the local optimum solution. The randomly chosen initial solution affects the quality of the generated solution. Therefore it is customary to generate several initial solutions randomly, say k of them, and repeat the iterative improvement procedure k times; the best of the k final solutions is taken as the optimal. The local search algorithm can be modified to start with a good initial solution which is obtained using a constructive procedure. For example, the farthest insertion algorithm or Christofides' algorithm can be used to generate an initial TSP tour. By introducing randomization in the constructive procedure, we can generate several good random solutions. If these initial solutions are sufficiently 'spaced apart' in the search space, then running the iterative improvement algorithm $k > 1$ times can be useful in searching for a better quality solution. However, since even a single execution of an iterative improvement algorithm can be expensive for large n, the above technique may require prohibitively large amounts of computer time. Parallel processing can be used to overcome this problem. In this section, we present our experiences in using parallel techniques to solve large travelling salesperson problems. An Intel iPSC/2 with 32 nodes[8] and an Alliant FX/80 with four advanced computational elements[9] were used to carry out the work described in this section.

Parallel farthest insertion heuristic

We assume an MIMD computational platform with P processors. Each processor reads in a copy the cost matrix C into its local memory. The file containing the cost matrix is stored on the concurrent file system (/cfs) so that all processors can have simultaneous access to the file. On a random basis, each processor selects a different city to start the tour construction. The procedure lrand48 is used for this purpose; this procedure is provided as a built-in function by the UNIX operating system, and returns a pseudo-random integer. The cities are numbered $0, 1, \ldots, n-1$ and each processor begins its tour with the city lrand48() modulo n. The procedure srand48 is used by each processor to initialize the pseudo-random sequence. In particular, processor i uses srand48(i + time) to initialize its sequence. The variable 'time' represents the 'seconds' portion of the Greenwich mean time at the instance when srand48 is invoked; it is obtained using the gmtime system call. Since each processor uses a personalized seed to initialize its pseudo-random sequence, it is highly unlikely that two or more processors start with the same city. It is easy to see that if two processors i and j use the farthest insertion heuristic starting with cities s_i and s_j, they will construct identical tours if $s_i = s_j$. Similarly, if $s_i \neq s_j$, the processors i and j will construct different tours. Let u be the number of unique starting cities when P processors are used to execute the parallel farthest insertion heuristic. Table 1 shows the results of running PFI on an iPSC/2 with 32 processors. Two problem instances were used and the experiment was repeated five times in each case. For each experiment, we show the minimum, maximum and average tour lengths found by the processors. As can be seen, u is sufficiently close to P and approaches P for larger values of n. This is important, because $P - u$ represents the amount of redundant work in the parallel farthest insertion algorithm. The redundancy can be further reduced by using better pseudo-random generators. The tour length obtained by a deterministic farthest insertion algorithm which always starts the tour with city 0 is shown as dfi. The PFI algorithm outperforms DFI in both average and best cases. For example, in problem p532, the best tour found by the PFI algorithm is 29 499, which is roughly 2% better than the tour length of 30 051 generated by DFI. This improvement can be further enhanced by making the following change to the PFI algorithm. Each processor executes the farthest insertion heuristic k times, starting with a random initial city. This amounts to Pk executions of the randomized farthest insertion procedure. We denote the modified PFI algorithm as multiple execution PFI, or MFI for short. The percentage improvements obtained by MFI for varying values of k are plotted in Figure 2.

Parallel version of Christofides' algorithm

A parallel algorithm to implement Christofides' approximation scheme was implemented on an Alliant FX/80 with four advanced computational elements. The main subroutines in the program are described below. Subroutine MINTREE finds the minimum spanning tree in the TSP input graph. The graph is specified in the form of an $n \times n$ matrix, C. Prim's algorithm is used to compute the

Table 1. Results of parallel farthest insertion heuristic for TSP

	Problem k24, n = 100, P = 32				
	Exp 1	Exp 2	Exp 3	Exp 4	Exp 5
u	20	23	24	23	22
max	24006	24089	24089	24089	24006
min	22047	21628	21628	21628	21628
avg	22738	22821	22761	22755	22650
dfi	23144				
	Problem p532, n = 532, P = 32				
	Exp 1	Exp 2	Exp 3	Exp 4	Exp 5
u	27	32	31	32	32
max	30874	30849	30629	30874	30849
min	29499	29576	29576	29499	29576
avg	30186	30193	30246	30158	30215
dfi	30051				

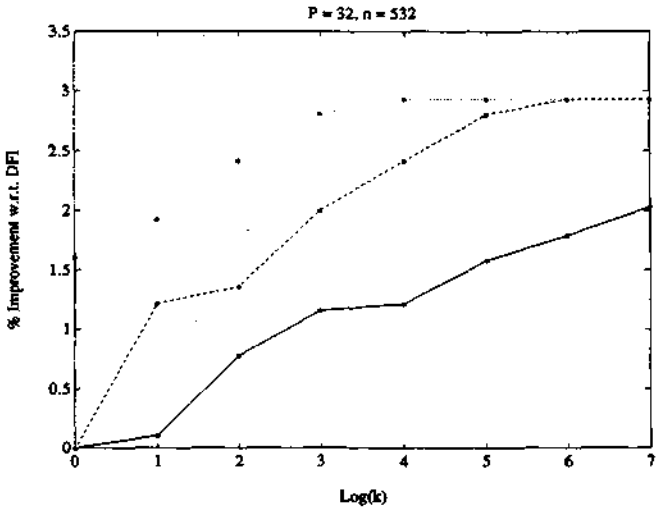

Figure 2. Results of multiple execution PFI for the 532-city problem. The dotted curve corresponds to P = 32; the dashed curve corresponds to P = 8; the solid curve corresponds to P = 1

minimum spanning tree[10]. Since there can be more than one minimum spanning tree in the graph, randomization can be introduced in the procedure by selecting the initial edge of the tree randomly. Subroutine PMATCH finds the minimum weight matching in a given graph. The algorithm due to Berkard and Derigs is used in implementing this subroutine[13]. Randomization can also be introduced in this algorithm, since more than one matching can satisfy the minimum weight property. Subroutine EULER is used in finding an Eulerean tour in a given graph. In all these subroutines, array representation is used to store a graph. For example, a graph $G = (V, E)$ on N nodes and M edges is represented by two integer arrays X and Y of length M; the entries $X(i)$, $Y(i)$ correspond to the two nodes that comprise edge i of the graph, $1 \leqslant i \leqslant M$. Such a representation leads to efficient vectorization as well as concurrentization on the Alliant FX/80. Figure 3 shows the speed up obtained when running the program on a consortium of k advanced computational elements, $1 \leqslant k \leqslant 4$.

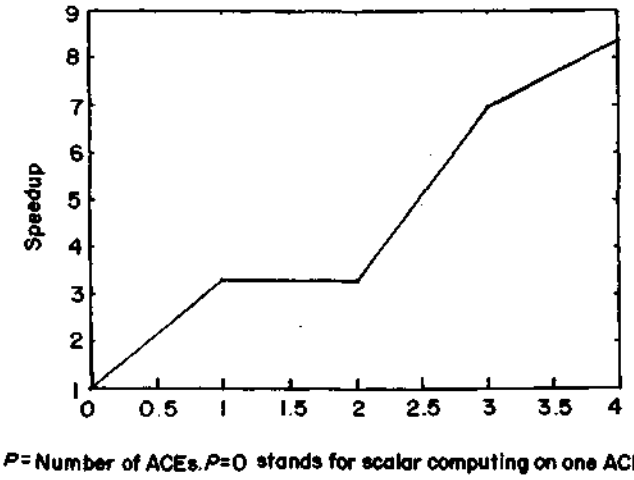

Figure 3. Performance of Christofides' algorithm on the Alliant FX/80

Parallel *k*-opt

Parallel two-opt and parallel three-opt algorithms were implemented on an iPSC/2. Several variations of the two-opt algorithm were implemented, as explained below.

(1) FI + P2OPT. Each processor obtains an initial tour using the farthest insertion heuristic. The initial city for farthest insertion is obtained using a pseudo-random generator, as discussed in the previous section. Each processor executes the 'two-opt' procedure starting with its initial tour; processors proceed independently of one another since each has a copy of the cost matrix. After all the processors converge to a local optimal tour, the shortest tour is identified using the global function gilow. This function applies a distributed algorithm to compute the minimum of P integers $x_1, x_2, \ldots, x_P$ as input to the function; processor i supplies the pointer to local variable x_i.

(2) RAND + P2OPT. Each processor applies the two-opt procedure to a randomly generated initial tour. A random tour T is generated by constructing a random permutation of $0 \ldots n-1$ as shown below. The two-opt procedure is identical to (1) above.

```
procedure RandomTour(n);
   for i = 0 to n - 1 do T[i] = i;
   for i = 0 to n - 1 do begin
      r1 = lrand48() mod n; /* Random integer, 0
           ... n - 1 */
      r2 = lrand48() mod n;
         swap (T[r1], T[r2]);
   end
end procedure;
```

(3) MFI + P2OPT. Identical to (1), except that each processor executes the farthest insertion technique k times and selects the best of the k tours as the initial tour. The parameter k is provided by the user as an input to the program.

(4) MFI + C2OPT. Each processor uses k executions of the farthest insertion heuristic to construct an initial tour. The best of these initial tours is identified using the gilow global function. One or more processors may have arrived at the best initial tour; among these, the processor whose id is the lowest broadcasts the best initial tour to all remaining processors. The following code is used to achieve the broadcast operation. Each processor sets a flag variable of 1 (or 0) if it has (or has not) found the best tour in the MFI phase. These flag variables are bunched together into a vector of P flags using the global collate operation called gcol. If i is the first non-zero entry in the flag vector, then processor i is the lowest indexed processor with the best tour from MFI.

```
/* cost contains the length of the best of k tours
constructed using the farthest insertion heuristic.
bestcost contains the smallest of cost1, ..., costP
and is calculated using gilow function.
*/
if (cost = bestcost) then flag = 1 else flag = 0;
   collate(flag, flagvector);
for i = 0 to P do
      if flagvector[i] = 1 then break;
if (mynode() = i) then
         broadcast(T)
```

```
    else
        receive (T);
```

The C2OPT phase of the algorithm is a distributed version of the 'two-opt' procedure. Recall that the two-opt procedure on an n city problem must evaluate $n(n-3)/2$ adjacent pairwise swaps during each iteration. The basic idea in parallelizing the procedure is to distribute these pairwise swaps equally among the P processors for concurrent evaluation. The following procedure shows the structure of the two-opt algorithm.

```
procedure two-opt;
begin
  /*
  let H = (x0, x1, ..., xn-1) be the n edges in the
  initial tour.
  the notation C(xi) is used to indicate the cost of
  edge xi.
  */
  repeat
    δmax = 0;
    for i = 0 to n - 3 do
      for j = i + 2 to n - 1 or n - 2 do
        /* Latter case for i = 0 only.
        Evaluate the cost improvement resulting
        from deleting edges xi, xj and adding
        new edges yp, yq which complete the
        tour */
        δmove = (C(xi) + C(xj)) - (C(yp) + C(yq));
        if δmove > δmax then begin
          save i and j;
          δmax = δmove
        end
    if δmax > 0 then
      update tour by deleting xi, xj and adding
      yp, yq;
  until δmax = 0;
end
```

The distributed two-opt algorithm is shown in Figure 4. The core computations are parallelized by distributing the $n-2$ iterations of the outer 'for' loop among the P processors. Thus each processor examines $n(n-3)/2P$ pairwise interchanges and identifies a pair (i, j) which results in the largest reduction in tour length; this largest reduction is denoted δ_{local_max}. At the end of the 'for' loop, the processors compute δ_{max}, the maximum of δ_{local_max}, in a distributed fashion using the gihigh global function. Several pairwise interchanges may result in a gain of δ_{max}; the smallest indexed processor which has such a pair broadcasts the interchange pair to all the processors. This enables all processors to update their copy of the tour. Therefore, the overhead of parallelization consists of

(a) computation of δ_{max} using gihigh
(b) broadcast operation explained above

Both these overheads require $O(\log P)$ time on a P-processor hypercube. When n is much larger than P, the core computations overshadow the overheads and a speed-up close to P is realized.

```
procedure distributed-two-opt;
begin
  repeat
    δlocal_max = 0; /* Best swap gain as seen by this processor */
    for i = mynode() to n - 3 step P do
          : /* core identical to two-opt */
    end
    δmax = max(P, k=1) (δlocal_max)
    if δmax > 0 then begin
      if mynode() is the smallest index such that
      (δlocal_max = δmax) then
        broadcast (swap pair (i, j), (p, q))
      else
        receive (swap pair (i, j), (p, q))
      update tour by deleting xi, xj and adding yp, yq;
    end
  until δmax = 0;
end
```

Figure 4. Distributed two-opt algorithm

The three-opt algorithm was parallelized in ways similar to the two-opt algorithm. Three-opt requires $O(n^3)$ computations per iteration and hence is computationally much more expensive than the two-opt. The core of the three-opt algorithm consists of three nested loops; the distributed version of three-opt, called C3OPT, distributes the n iterations of the outermost loop among P processors for concurrent execution in a manner similar to C2OPT.

The different variations of the two-opt algorithm were tested against standard benchmark problems. For the famous 532-city benchmark, the shortest tour of length 28 285 was generated by the P3OPT algorithm. 32 processors were used in the experiment and the algorithm required more than three days to converge.

EXPERIMENTAL RESULTS

Table 2 shows the results of parallel search techniques when applied to several instances of TSP. P3OPT generated the best solution in each case; it was also the most compute-intensive algorithm. The tour length shown against an algorithm A corresponds to the minimum length tour generated using A, i.e. the minimum taken over all experiments using A with different values for the number of processors P. Further, the same algorithm A may generate its best solution several times, for different values of P; the execution time shown in the table against A is the minimum over all such cases.

The performance of FI + P2OPT as a function of P is shown in Figure 5 for two problems, the 532 city problem (p532) and a 100 city problem (k24). The performance of the RAND + P2OPT technique is shown in Figure 6 for the 532-city problem. It is seen that with an increase in the number of processors, both algorithms give better results in terms of solution quality. However, the synchronization overheads are also higher when there are more processors, resulting in an increase in execution time. For the same value of P, FI + P2OPT always performed faster than RAND + P2OPT. The solution qualities of the two algorithms were compared for the same values of P and $k = 1$. It was also observed that FI + P2OPT consistently generated better quality solutions. This conforms with the folklore that a constructive initial solution is better than a random initial solution. To test if MFI + P2OPT outperforms FI + P2OPT, the two algorithms were compared with identical inputs and were executed with the same

Table 2. Results of parallel search for TSP. All times are in seconds, except for the last entry in the P3OPT column

Problem name	No of cities	PFI		FI + P2OPT		FI + P3OPT		MFI + C2OPT	
		Tour	Time	Tour	Time	Tour	Time	Tour	Time
k24	100	21628	1.68	21591	2.59	21247	9060	21628	0.98
k25	100	22926	1.86	22587	2.48	22312	7108	22640	3.25
k26	100	20878	1.67	20865	2.39	20703	5388	20878	0.99
k27	100	21868	1.99	21594	2.33	21347	8403	21731	1.26
k28	100	22498	1.99	22441	2.29	22062	9285	22456	1.41
p532	532	29576	32.9	29405	96.9	28285	3 days	29195	103.8

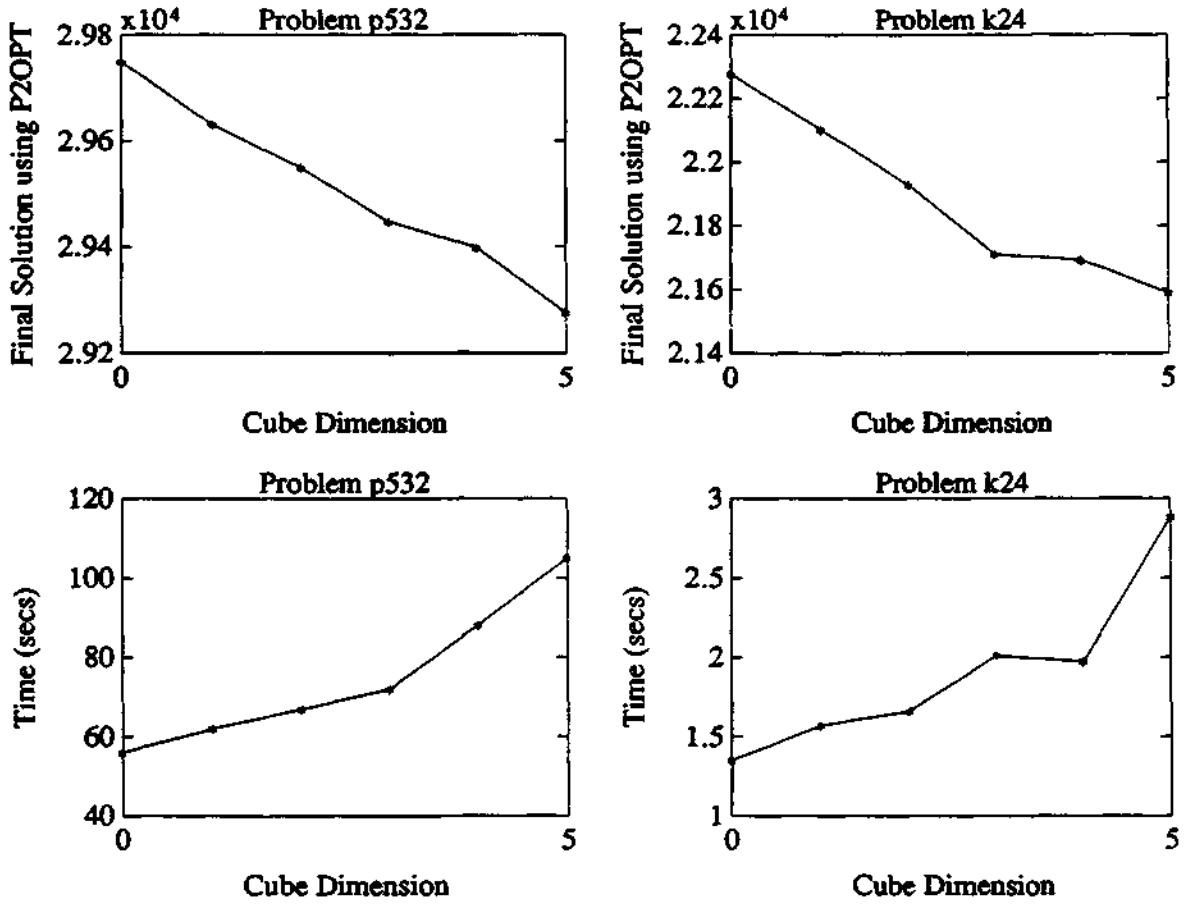

Figure 5. Performance of FI + P2OPT as a function of the number of processors

number of processors. Figure 7 shows the results obtained using MFI + P2OPT. For a given number of processors, increasing k improves the solution quality up to a point: for larger values of k, a saturation effect sets in. For example, when $P = 16$, $k > 16$ does not lead to better quality solutions.

Figure 8 shows the performance of MFI + C2OPT. Four different plots are shown, corresponding to $P = 2, 8, 16, 32$. With increase in k, the speed up of the parallel algorithm is seen to improve. For $k \geq 128$, we observed that speed up reaches the optimum value of P.

Figure 9 shows graphically the tours obtained through the use of four different techniques: Christofides' algorithm, the two-opt and three-opt algorithms initiated with a tour obtained by Farthest Insertion, and finally the two-opt algorithm initiated with a random tour. The 532 city problem from Reference 3 is used in this comparison. As can be seen, Christofides' algorithm generates the longest tour, and the three-opt algorithm generates the shortest possible tour (5% longer than the best known tour of 27 686). In this comparison, the best solution generated by multiple runs of each algorithm was used.

CONCLUSIONS

In this paper, parallel optimization techniques were described for the travelling salesperson problem. When applied to computationally hard optimization problems, parallel processing techniques can influence both the

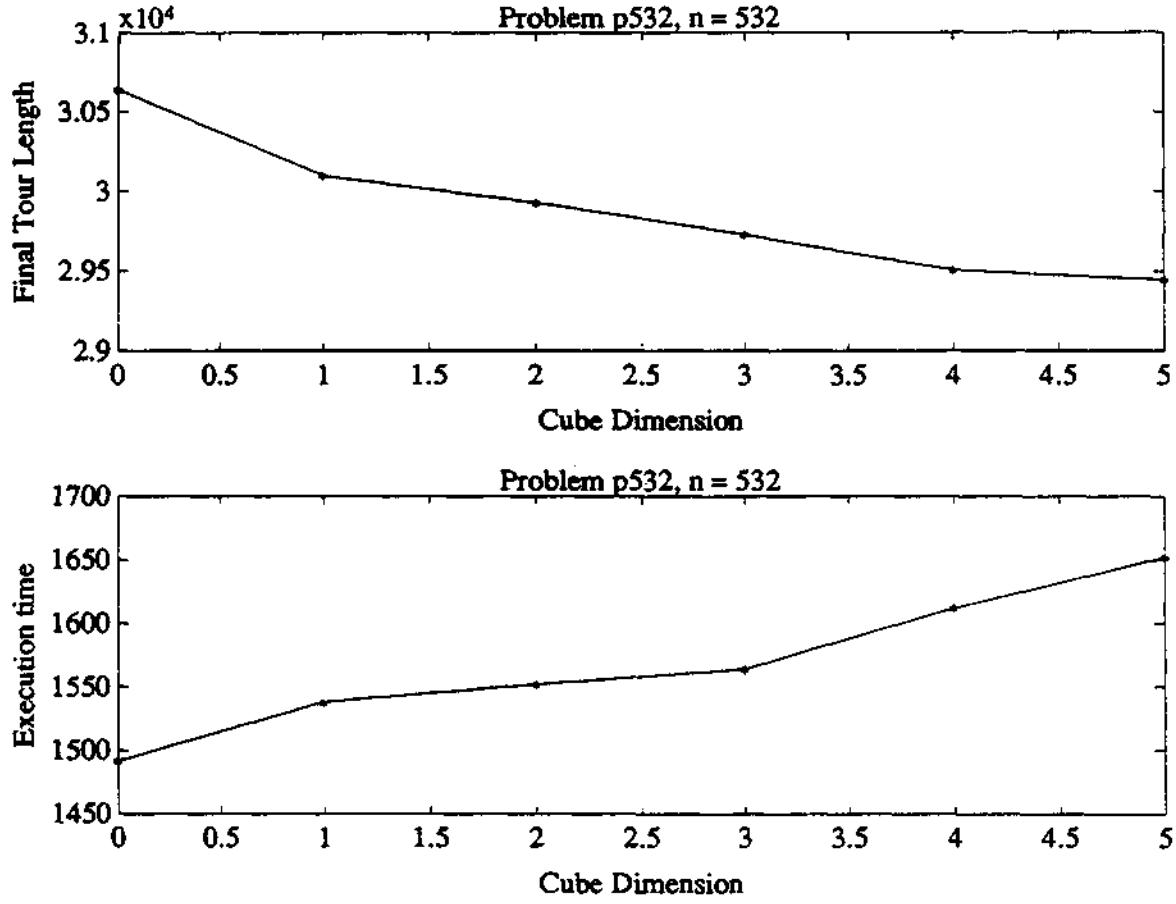

Figure 6. Performance of RAND + P2OPT for the 532 city problem

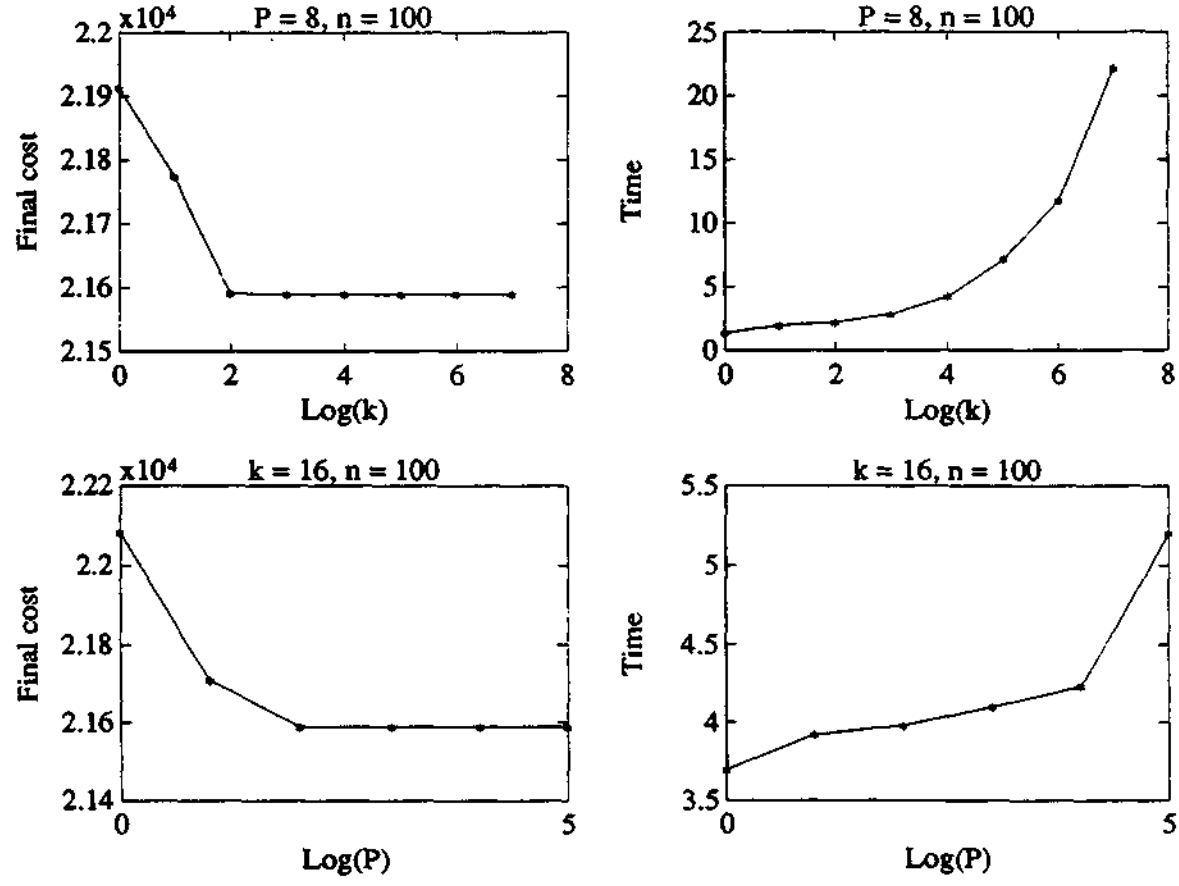

Figure 7. Performance of MFI + P2OPT for the 100 city problem

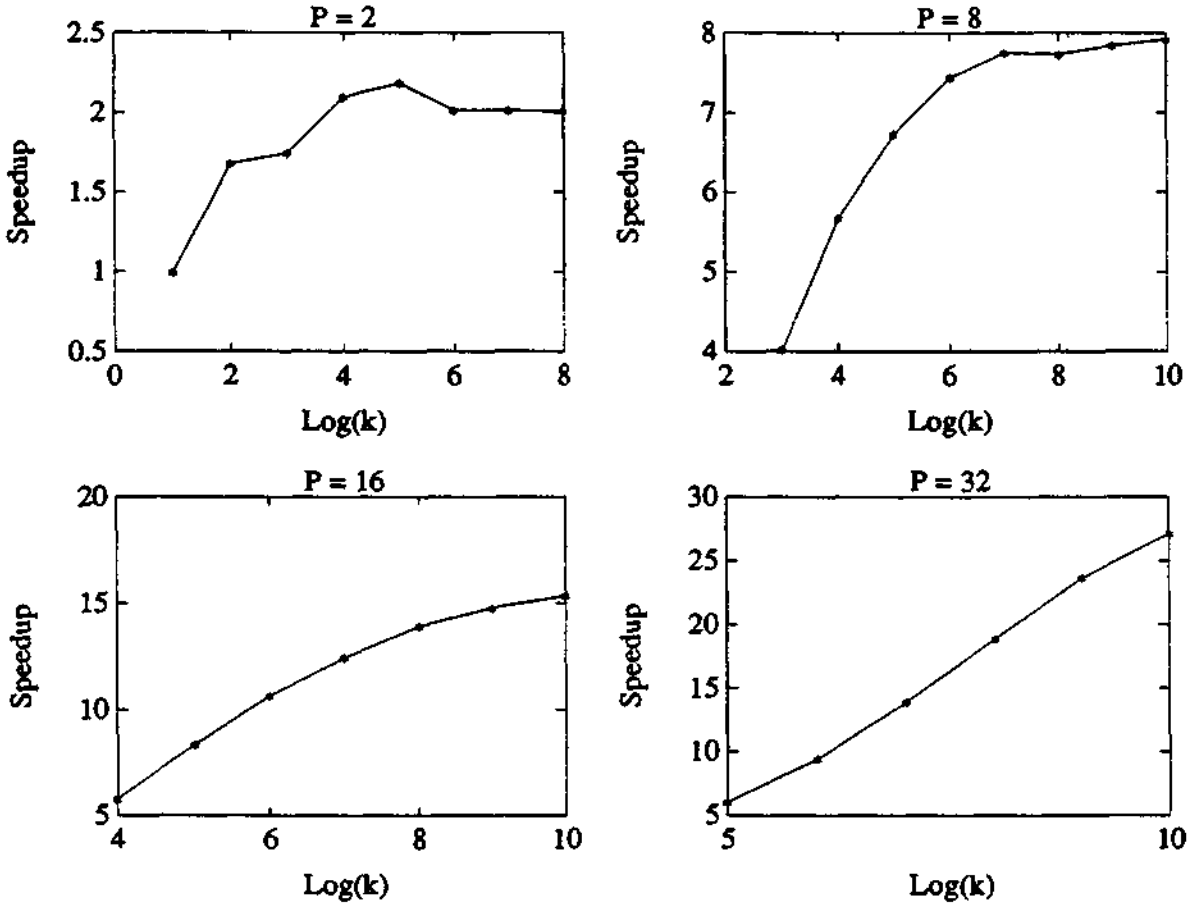

Figure 8. Performance of MFI + C2OPT for the 532 city problem

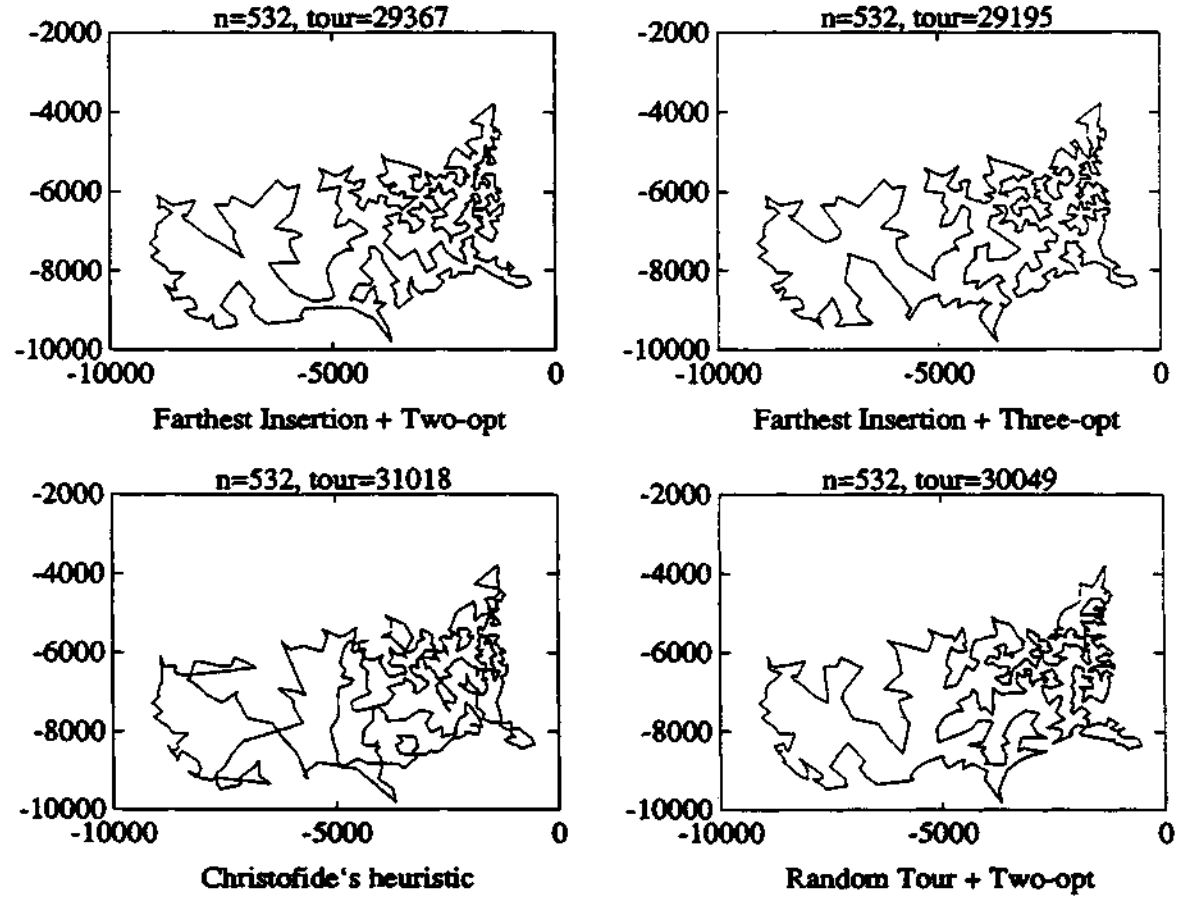

Figure 9. Performance of Christofides' algorithm on an Alliant FX/80

speed up and quality of the final solution. We have developed parallel algorithms for the TSP on two different computational platforms: the Intel iPSC/2 and the Alliant FX/80. In both cases, speed ups were close to the number of processors used. It was experimentally demonstrated that better quality solutions can be obtained through parallel randomization. Multiple processors are used to explore the combinatorial search space starting at different initial states. Presently, there is no way to guarantee that the state subspaces searched by two different processors are indeed non-overlapping. This property is desirable since it would ensure that the parallel algorithm necessarily generates better quality solutions compared to its sequential counterpart. Improvements to this work will come from parallel optimization algorithms that ensure the non-overlap property. The parallel algorithms described in this paper are well suited for implementation on MIMD computers. The advantage of algorithms such as PFI, P2OPT and P3OPT is that their existing sequential implementation can be ported to a parallel machine without much effort. Distributed algorithms such as C2OPT and C3OPT require more elaborate parallel implementation.

The techniques described here are also sufficiently general. For example, iterative improvement algorithms similar to 'two-opt' are used for circuit placement and routing. Goto[12] describes a λ-opt algorithm for two-dimensional circuit placement which starts with a good random initial placement and improves it by shuffling λ cells at a time. Brouwer and Banerjee[11] have described a two-layer channel routing algorithm which starts with a track assignment using exactly α tracks, α being the channel density of the problem. The initial track assignment may violate vertical constraints. A move consists of altering the track assignment for two nets. The algorithm performs moves and accepts those that decrease the number of vertical constraint violations. The algorithms in References 11 and 12 and many others can be adapted for MIMD implementation using our techniques.

REFERENCES

1 **Lin, S and Kernighan, B W** 'An effective heuristic algorithm for the travelling-salesman problem' *Oper. Res.* Vol 21 (1973) pp 498–516

2 **Kirkpatrick, S, Gelatt, C D and Vecchi, M P** 'Optimization by simulated annealing' *Science* Vol 220 No 4598 (May 13 1983) pp 671–680

3 **Padberg, M and Rinaldi, G** 'Optimization of a 532-city symmetric traveling salesman problem by branch and cut' *Oper. Res. Lett.* Vol 6 No 1 (1987) pp 1–7

4 **Litke, J D** 'Improved solution to the traveling salesman problem with thousands of nodes' *Comm. ACM.* (December 1984) pp 1227–1231

5 **Chan, D and Mercier, D** 'IC insertion: an application of the travelling salesman problem' *Int. J. Prod. Res.* Vol 27 No 10 (1989) pp 1837–1841

6 **Golden, B *et al.*** 'Approximate traveling salesman algorithms' *Oper. Res.* Vol 28 (1980) pp 694–711

7 **Lawler, E L *et al.*** (Eds) *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization* John Wiley, New York, USA (1985)

8 **Pase, D M and Larrabee, A R** 'Intel iPSC concurrent computer' in **Babb II, R G** (Ed) *Programming Parallel Processors* Addison-Wesley, Reading, MA, USA (1987)

9 *FX/Fortran Programmer's Manual* Alliant Computer Systems Corp (1987)

10 **Aho, A V, Hopcroft, J E and Ullman, J D** *Data Structures and Algorithms* Addison-Wesley, Reading, MA, USA (1983)

11 **Brouwer, R J and Banerjee, P** 'A parallel simulated annealing algorithm for channel routing on a hypercube multiprocessor' in *Proceedings of the International Conference on Computer Design* (1988) pp 4–7

12 **Goto, S** 'An efficient algorithm for the two-dimensional placement problem in electrical circuit layout' *IEEE Trans. Circ. Syst.* Vol CAS-28 (1981) pp 12–18

13 **Burkard, R E and Derigs, U** *Assignment and Matching Problems: Solution Methods with FORTRAN Programs* Springer-Verlag, Berlin (1980)

C.P. Ravikumar obtained a Bachelor's degree in electronics from Bangalore University, India, in 1983, and a Master's degree in computer science from the Indian Institute of Science in 1987. He obtained his PhD in computer engineering from the University of Southern California in 1991, and is presently an Assistant Professor in the Department of Electrical Engineering at the Indian Institute of Technology, Delhi. During 1984–1985 and in 1987, he worked as a software engineer. His chief areas of interest are VLSI design, design automation for VLSI, parallel and distributed computing, computer architecture and graph algorithms.