Distributed Simulated Annealing Algorithms for Job Shop Scheduling

K. Krishna, K. Ganeshan, and D. Janaki Ram

Abstract—Job shop scheduling belongs to the class of NP-hard problems. There are a number of algorithms in literature for finding near optimal solution for the job shop scheduling problem. Many of these algorithms exploit the problem specific information and hence are less general. However, simulated annealing algorithm for job shop scheduling is general and produces better results in comparison with other similar algorithms. But one of the major drawbacks of the algorithm is that the execution time is high. This makes the algorithm inapplicable to large scale problems. One possible approach to reduce the execution time of the algorithm is to develop distributed algorithms for simulated annealing. In this paper, we discuss approaches to develop distributed algorithms for simulated annealing for solving the job shop scheduling problem. Three different algorithms have been developed. These are the Temperature Modifier, the Locking Edges and the Modified Locking Edges algorithms. These algorithms have been implemented on the Distributed Task Sharing System (DTSSS) running on a network of 18 sun workstations. The observed performance showed that each of these algorithms performs well depending on the problem size.

I. INTRODUCTION

Job Shop Scheduling (JSS) belongs to the class of NP-hard optimization problem. Even in the NP-hard class of problems, JSS appears to belong to the more difficult ones [1]. The JSS problem can be described as follows. A set of jobs whose operations are to be processed on a set of machines is given. Each job consists of a sequence of operations. Each of these operations have to be processed uninterrupted on a given machine for a specified length of time. There is an additional constraint that each machine can process at most one operation at a time. A schedule is an allocation of the operations to time intervals on the machines. The problem is to find the schedule of minimum time. Solving the JSS problem requires a high computational effort and considerable sophistication. A much simpler task is to find a reasonably good schedule not necessarily the optimum one. A number of algorithms have been developed to address this simpler task. They include the Giffler and Thompson algorithm [2], the shifting bottleneck algorithm [3], the simulated annealing algorithm [1], [4], etc. Computational results show that simulated annealing can find shorter makespans than the other recent approximation algorithms [1]. This is however at the cost of large execution times.

Attempts to reduce the execution time taken by the simulated annealing algorithm required that it be distributed. One approach to distribution is to divide the problem space and then distribute the problem space to various nodes in a distributed network. The other approach is to change the temperature modifier parameter. In this approach the algorithm is run on various nodes of the network on the whole problem space. However the size of the steps at which the temperature is decreased is raised so that the time taken by the algorithm on each node is less. The effect of raising the temperature modifier is that the probability of the algorithm getting stuck at a local minimum is high. However the probability that two nodes will get stuck at the same local minimum is less. Hence the minimum obtained among all the nodes will match the global minimum.

In this paper we discuss both these approaches. We also show that depending on the problem size, one of these approaches can be selected.

II. OVERVIEW

In this section a brief overview of the problem, Simulated Annealing and Distributed Algorithms has been provided.

A. The Problem

The problem can be stated mathematically in the following fashion. Given a set $A$ of $n$ jobs, a set $B$ of $m$ machines and a set $C$ of $N$ operations, $\max_{v \in C} s_v + t_v$ is minimized subject to

$$\begin{align*}
 s_v &\geq 0 \text{ for all } v \in C, \\
 s_w - s_v &\geq t_v \text{ if } v \text{ precedes } w; \quad v, w \in C, \\
 s_w - s_v &\geq t_v \forall s_w - s_v \geq t_w \text{ if } m_v = m_w, \quad v, w \in C,
\end{align*}$$

where $v, w$ are any operations belonging to the set $C$

$s_v =$ start time of the operation $v$

$t_v =$ processing time of the operation $v$

$m_v =$ machine on which the operation $v$ is performed.

The following assumptions are made for the problem:
1) All the jobs are available at time zero.
2) There are no machine breakdowns.
3) Operation times of the jobs on the machines are known beforehand.

Manuscript received February 8, 1993; revised August 27, 1993, and August 20, 1994.

K. Krishna is with Nucleus Software, Madias 600 036, India.

K. Ganeshan is with Wipro Infotech, Bangalore 560 001, India.

D. Janaki Ram is with the Department of Computer Science and Engineering, Indian Institute of Technology, Madras 600 036, India.

IEEE Log Number 9410812.
A disjunctive graph model \((G)\) with a set of vertices \((V)\), a set of arcs \((A)\) and a set of edges \((E)\) can be used for representing the problem [1]. The disjunctive graph \(G = (V, A, E)\) is defined as follows:

The set of vertices \(V\) consists of all the vertices in \(C\) and two vertices numbered 0 and \(N + 1\) representing the fictitious start and end operations respectively. The processing time of the operation is denoted as the weight of the vertex. The two fictitious operations 0 and \(N + 1\) have operation times of zero.

The set \(A\) contains arcs connecting consecutive operations of the same job, as well as arcs from 0 to first operation of each job and from the last operation of each job to \(N + 1\).

The edges in the set \(E\) connect operations to be processed by the same machine.

This is illustrated by taking example 1 (refer to Table 1). The example problem can be represented as a disjunctive digraph (refer to Fig. 1(a)). The set of vertices, arcs and edges in this case are \(\{0, 1, 2, 3, 4 \ldots 17\}, \{(0, 1), (0, 5), (0, 9)\}, \{(0, 13), (1, 2)\}, \ldots \) and \(\{(5, 1), (6, 2), (15, 5)\}, \ldots \) respectively.

The directed arcs in the figure denote that the operations are to be processed in that order. The edges in the graph have two possible orientations. For example, if \((v, w)\in E\), then the edge can be directed from either \((v, w)\) or \((w, v)\). The assignment of orientations to these edges forms a schedule. This schedule will indicate the complete sequence of operations to be carried out on each machine. The longest path in this digraph called the ‘makespan’ gives the total time for completion of the operations of all the jobs. The objective will be to assign orientations to the edges such that the makespan is minimum. The orientations of the edges decide the sequence of operations performed on one machine. This is equivalent to finding the permutations of the operations on each machine to minimize the makespan. This problem is one of the hardest combinatorial optimization problems. To solve this problem, a heuristic method called the Simulated Annealing (SA) is employed.

**B. Simulated Annealing (SA)**

SA belongs to the type of local search algorithms [5]. The algorithm chooses an initial solution at random. A neighbor of this solution is then generated by a suitable mechanism and the change in the cost function of the neighbor is calculated. If a reduction in the cost function is obtained, the current solution is replaced by the generated neighbor. On the other hand, if the cost function \(f\) of the neighbor is more, the generated neighbor replaces the current solution with an acceptance probability function given by: \(\text{EXP}(-\{f[j] - f[i]\}/T)\) where \(f[i]\) and \(f[j]\) are the cost functions of the generated state and the present state respectively. \(T\) is a control parameter which corresponds to the temperature in the physical annealing process. The above acceptance function implies that small increases in \(f\) are more likely to be accepted than large increases, and also that when \(T\) is high, most of the generated neighbors are accepted. However, as \(T\) approaches zero most of the cost increasing transitions are rejected. The initial temperature in the simulated annealing algorithm is kept high such that the algorithm does not get trapped in a local minimum. The algorithm proceeds by...
generating a certain number of neighbors at each temperature, while the temperature parameter is gradually dropped. This algorithm leads to a near optimal solution.

C. Distributed Algorithms

Distributed algorithms represent the algorithmic formulation of Distributed Problem Solving (DPS) [6]–[9]. DPS can be termed as cooperative problem solving by a loosely coupled network of problem solvers. The main purpose of distributed algorithms is to exploit the processing power of a number of nodes on a network. Since SA technique for JSS is inherently sequential and highly compute intensive, distributed algorithms for SA technique for JSS can make the technique applicable for large scale problems. Two different approaches to the development of distributed algorithms for SA technique for JSS have been contemplated. One approach is to divide the problem space into independent subtasks. The algorithm based on this approach is termed as the Locking Edge algorithm. This algorithm is modified for large problem sizes. The other approach involves distributing the reduction rate of the temperature among various nodes of the network. The algorithm based on this approach is called the Temperature Modifier algorithm. These algorithms have been explained in subsequent sections.

III. DISTRIBUTED ALGORITHMS FOR JOB SHOP SCHEDULING

This section describes the development of distributed algorithms for JSS using SA technique.

Initially a sequential algorithm [1], [5] is presented which will be modified subsequently to develop distributed algorithms.

A. Sequential Algorithm

The sequential algorithm involves the following major steps: 1) Finding an initial schedule, 2) Evaluating cost of the schedule, 3) Finding the critical path, 4) Generating a neighbor. These are discussed in detail below.

1) Initial Schedule: Given a disjunctive graph \( G = (V, A, E) \) for solving the problem, an initial schedule is generated. The Gifford and Thompson algorithm [2] is employed for this purpose. The algorithm attempts to construct the schedule by considering all the operations \( (n) \) on all the machines \( (m) \). The criteria employed being the earliest starting time and the processing time of each of the operations. At each stage an operation not yet included in the schedule and requiring minimum time is chosen and included in the partial schedule. The partial schedule becomes a complete schedule when all the operations of the jobs are included in the schedule. The generated schedule can be represented as a digraph.

2) Cost Function: After obtaining the digraph representing an initial schedule, the earliest and the latest start times of each of the operations in the graph are calculated. The Critical Path Method (CPM) is used for this purpose. The makespan is the earliest start time or the latest start time of the last operation. This forms the cost of the schedule.

3) Critical Path: After evaluating the cost function, the critical path in the digraph is identified. The critical path can be defined as a set of edges from the first vertex to the last vertex which satisfy the following properties:

a) The latest start time and the earliest start time of each vertex on the edge must be the same.

b) For the same edge \( u \rightarrow v \), the sum of the start time and the operation time of \( u \) must be equal to the start time of \( v \).

An edge in the critical path is reversed to generate a neighbor and this is discussed in the next section.

4) Generating a Neighbor: The neighborhood of a schedule can be defined as a set of schedules that can be obtained by applying the transition function on the given schedule. Neighborhoods are usually considered by first choosing a simple transition function. A transition in the case of JSS problem is generated by choosing the vertices \( v \) and \( w \) such that [1]:

a) \( v \) and \( w \) are any two successive operations performed on the same machine \( k \);

b) \((v, w)\in E\) is a critical edge, i.e. \((v, w)\) is on the longest path of the digraph.

A neighbor is generated by reversing the order in which \( v \) and \( w \) are processed on the machine \( k \). It has been shown that by using this transition function it will be possible to eliminate infeasible solutions and also keep non-decreasing paths out of the search space [1].

Thus, in the digraph such a transition results in reversing the edge connecting \( v \) and \( w \) and replacing the edges \((u, v)\) and \((w, x)\) by \((u, w)\) and \((v, x)\) respectively, where \( u \) is the previous operation to \( v \) on the same machine, and \( x \) is the next operation to \( w \) on the same machine.

B. Distributed Algorithms

We have developed three distributed algorithms for JSS by modifying the sequential algorithm. They can be termed as:

1) Temperature Modifier Algorithm,

2) Locking Edges Algorithm,

3) Modified Locking Edges Algorithm.

They are explained in detail below.

1) Temperature Modifier Algorithm (TMA): The choice of the temperature modifier in the sequential algorithm affects the probability of the algorithm getting stuck at a local minimum. A low value for the modifier makes the algorithm fast but the probability of the algorithm getting stuck at a local minimum is high. To reduce this probability, the sequential algorithm can be simultaneously executed on different nodes. The Temperature Modifier Algorithm is briefly stated in Table II.

2) Locking Edges Algorithm (LEA): It can be observed that the Temperature Modifier algorithm does not result in much improvement with respect to the execution time on the computer. To improve this, Locking Edges algorithm has been developed. This algorithm generates subtasks by ‘locking’ edges in the digraph. The term ‘locking’ can be defined as marking an edge of the digraph such that its orientation cannot be changed. If the number of edges locked in the digraph is \( m \), then we can generate \( 3^m \) equal subtasks. This is equivalent to dividing the entire search space into \( 3^m \) divisions. For
TABLE II
TEMPERATURE MODIFIER ALGORITHM

1. Select appropriate temperature modifiers for different nodes.
2. Run the following sequential algorithm with the chosen temperature modifier on each node.
3(a) Generate the initial schedule, given the processing times of all operations and the machine order for each job.
3(b) Repeat
   - counter = 0;
   - Compute the cost of the initial schedule \( f(i) \);
   - Repeat
     - Calculate the critical path;
     - Generate the neighborhood;
     - Compute the cost of the generated schedule \( f(i+1) \);
     - Accept or reject the generated schedule with the probability \( \exp [-\frac{f(i+1)-f(i)}{T}] \);
   - until (counter = number of runs at a temperature);
   - \( T = T \times \text{modifier} \);
   - until (\( T = 0 \) or Minimum schedule not changed for a long time);
3(c) Send the result back to the central node.

---

example, if the number of locked edges in the digraph is one, then we can generate 3 subtasks. These subtasks can be obtained as follows:

- a) In the first subtask, the search space consists of all possible orientations of the edge except that of the locked edge. The orientation of the locked edge remains intact.
- b) In the second subtask, the orientation of the locked edge is reversed.
- c) In the third subtask, the locked edge is removed.

The generation of three sub-tasks are illustrated by taking example 1 (please refer to Table I). The initial schedule generated for the example problem is represented in Fig. 1(a). In this schedule the edge 15-5 is locked to generate the sub-tasks. Hence the initial schedule in Fig. 1(a) with edge 15-5 locked forms the starting schedule for sub-task 1. The starting schedule for the second sub-task is generated by reversing the locked edge 15-5 in the initial schedule and the schedule thus obtained is given in Fig. 1(b). The starting schedule for the sub-task 3 is generated by removing the locked edge 15-5 from the initial schedule and this is shown in Fig. 1(c).

The above concept can be further extended to a case in which there can be \( m \) locked edges. Fig. 2(a)–(c) explains the method of generation of the subtasks. The generated subtasks are assigned to the cooperating nodes in the Distributed Problem Solving network. The detailed algorithm is presented in Table III.

The edges to be selected for locking are chosen at random. The edge chosen affects the division of the search space and it influences the quality of the solution in some cases. This is discussed in the next section.

3) Modified Locking Edges Algorithm (MLEA): It is observed that as more and more edges are locked, it resulted in a decrease in the performance with respect to the optimal scheduling cost in some cases. This is explained theoretically in subsequent paragraphs.

A new term called 'Collision' is defined in the case of locking edges version of the distributed algorithm. In locking edges version, the search space is divided equally among all the cooperating nodes. For a 3 node distribution, the search space can be represented as in Fig. 3(a) and for a 9 node version it can be represented as in Fig. 3(b). One of the edges on the critical path is chosen at random for generating a neighbor. If this selected edge (say \( e1 \)) affects the locked edge (say \( e2 \)), then such as situation is termed as 'collision of edge \( e1 \) with locked edge \( e2 \). In such case the edge \( e1 \) is rejected and a new edge is selected at random to generate a neighbor. It can be interpreted as a node trying to 'penetrate' into the search space belonging to the other node. It can be observed from the figures that these collisions will be more in the 9 node case compared to that in the 3 node case. This results in a marginal increase in the schedule cost as compared to the previous case. In cases where the solution corresponding to

---

TABLE III
LOCKING EDGES ALGORITHM

1. Generate the initial schedule with the input given.
2. Choose the number of edges to be locked, say \( m \).
3. Generate the subtasks depending on the number of locked edges. i.e., for \( m \) locked edges \( 2^m \) subtasks are generated.
4. Assign each of the subtasks to the cooperating nodes. Pass information about the locked edges.
5. Wait for the results from the cooperating nodes.
6. Choose the optimal cost solution.
TABLE IV
MODIFIED LOCKING EDGES ALGORITHM

1. Generate the initial schedule with the input given.
2. Choose the number of edges to be locked, say n.
3. Generate the first set of subtasks \( \{ 3 \text{ power } n \} \).
4. Assign the subtasks to the cooperating nodes. Pass information about the locked edges also.
5. Generate the second set of subtasks \( \{ 3 \text{ power } n \} \).
6. Assign the subtasks to the cooperating nodes. Pass information about the locked edges also.
7. Wait for the results from the cooperating nodes.
8. Choose the optimal cost solution.

The minimum exists on the boundary of the search spaces of two nodes, there is less likelihood of it being reached in the 9 node case because of collisions.

To minimize the affect of collisions, a modification to the ‘locking edges’ version is attempted. This can be explained as follows. Consider the case where a single edge is locked. Here there are two sets of nodes. The first set of nodes consists of three nodes and will be assigned the tasks same as in the ‘locking edges’ case. The second set of nodes also consisting of three nodes are assigned tasks in such a way that the boundary points of the previous set of search spaces become the active search spaces for these nodes. As pointed out previously, this is done mainly to reduce the effect of collisions. This can be extended to any number of locked edges. The detailed algorithm is given in Table IV.

The edges to be locked are selected at random and the selected edges might be the cause for collisions. However it is not easy to predict beforehand this effect of a particular edge on collisions. Hence in the absence of such knowledge, the modified locking edge version guarantees that the search space is divided to avoid excessive collisions at least in one set of search space division.

IV. IMPLEMENTATION

These algorithms have been implemented on the Distributed Task Sharing System (DTSS) [7] running on a network of Sun Workstations having three servers of Sun 3/60 and 15 clients of Sun 3/50 connected together by a thin ETHERNET. The DTSS has been developed around a message kernel. The message kernel is implemented using datagram sockets. Messages across nodes are transferred by these datagrams. The message kernel provides support for reconfiguring the nodes on the network, for sending and receiving the task award and the result messages. The nodes on the network are initially configured such that one of the nodes is identified as a central node. During the initial configuration as many other required client nodes are also identified. The central node has the responsibility for dividing the search space and for communicating the tasks to the client nodes through task award messages. The client nodes after receiving the task award messages, execute the required task and send back the results through the result messages. The network is highly flexible and can be reconfigured with any number of client nodes.

The messages are retransmitted in the case of loss of message packets in transmission. This is detected by the nonreceipt of an acknowledgement for the packets sent within a specified time-out period. The node failures are also detected in a similar fashion. In case of node failures, the corresponding sub-tasks are assigned to other client nodes available on the network.

Before proceeding further on the implementation details, two terms are defined:

Central Node: This node holds the responsibility of task division, task award to client nodes, receiving result messages and synthesis of the final solution from the results obtained from the client nodes.

Client Node: This node solves the subtask assigned to it and returns back the result.

A. Temperature Modifier Algorithm

The implementation of the algorithms has been carried out by a central node and a set of client nodes. The central node generates the initial schedule given the processing times of all the operations and the machine order for each job. This node then sends the initial schedule and also different temperature modifier parameters to each of the client nodes on the network. The client nodes execute the sequential algorithm (Table II) with their corresponding temperature modifiers and send back the result to the central node.

The central node after receiving the results from the client nodes, chooses the solution having minimal cost.

The quality of the solution generated is influenced by the cooling rate. We employ a three parameter cooling schedule same as in [1]. The parameter delta controls the rate of cooling. A lower value of delta reflects a slower cooling rate and consequently the algorithm takes more time. The value of delta employed is in the range of \(10^{-7}\text{–}10^{-5}\). Hence choosing an appropriate value of delta for each node in the network is required. A marginally high value for delta than that employed in the sequential algorithm can be chosen for TMA. For example, if a delta value of \(10^{-2}\) is employed for a sequential algorithm, for a TMA case, a delta of \(0.5 \times 10^{-1}\) may be chosen. As a higher cooling rate affects the quality of the solution, it may be appropriate to employ TMA only in small size problems, where LEA or MLEA doesn’t give better results.
B. Locking Edges Algorithm

Similar to the above implementation, a central node and a set of client nodes participate in the execution of the problem. In this case, the number of client nodes is equal to the number of generated subtasks. The generated subtasks are based on the number of locked edges.

Initially, the central node generates the initial schedule and depending on the number of locked edges it generates the subtasks and assigns each one to one of the client nodes. The client nodes execute the subtask and return the result back to the central node. The central node synthesizes all the results and chooses the minimum cost solution as the best one.

C. Modified Locking Edges Algorithm

To implement this algorithm, a central node and sets of client nodes are identified. The central node divides the search space into sub-tasks and assigns them to one set of client nodes. This process is same as the one described in Locking Edges algorithm. However, in the case of Modified Locking Edges algorithm, the search space is again divided by the central node such that the boundaries of the search space in the earlier set become the active search spaces in this case. The sub-tasks generated in this process are assigned by the central node to the next set of client nodes. Thus many sets of client nodes participate in the problem solving in this case.

V. RESULTS AND OBSERVATIONS

A. Comparison of Results of Sequential Implementation

We attempted comparison of the results of our implementation of the sequential algorithm with that of the VAL (Lar Vaanhoven, Aarts, and Lenstra) implementation. We have considered the same three problem instances FIS1, FIS2 and FIS3 from Fischer and Thompson [10]. FIS2 is one of the difficult test cases and is stated that it defined solution to optimality for more than twenty years [1]. It has been shown that sequential simulated annealing produces better results compared to ABZ (Adams, Balas and Zawack) method [3] and MSS (Matsuo, Suh and Sullivan) method [4] but this is at the cost of large running times [1].

Table V compares the results of our implementation with that of the VAL implementation. We have considered the three parameter cooling schedule same as in the VAL implementation. The parameter delta controls the cooling rate. A lower value for delta means a lower cooling rate. The results of our sequential implementation are marginally better than VAL implementation. Our observations regarding sequential implementation of the algorithm are summarized below:

1) The initial and final temperatures for different problem sizes cannot be the same. We found that for higher problem sizes, it may be advantageous to keep the initial temperature high as the search space is more in this case.

2) The number of neighbors generated at any temperature also varies depending on the problem size. However, the bound \( m \times n = m \times n \) (\( m \) is the number of machines and \( n \) is the number of jobs) on the neighbors generated at a temperature appears inappropriate for small size problems. This bound is not able to generate a markov chain of sufficient length at a given temperature for convergence to global optimum.

B. Discussion of Implementation Results of the Distributed Algorithms:

As stated earlier, one of the major drawbacks of the sequential SA algorithm is the large execution times taken by it. The main purpose of developing the distributed algorithms is to reduce the execution times.

We compare the distributed algorithms with the sequential algorithms based on the improvement in the execution times and the quality of the solution obtained.

We have taken several instances of the job shop scheduling problem by varying the size of number of jobs and number of machines. These instances of the problem are generated using an algorithm which assigns operations times from 1-90 to the operations at random.

The performance of the various distributed algorithms for these problem instances is tabulated in Table VI. The three problem instances from Fischer and Thompson have also been implemented and the performance of these problem instances has been tabulated in Table VII. Graph 1 (Fig. 4) shows the plot of problem size versus execution time and Graph 2 (Fig. 5) shows the plot between the problem size and the percentage deviation from the optimal solution cost.

We summarize our results of the distributed algorithms below:

1) Distributed algorithms generally performed well from the view point of execution time. LEA performed extremely well and gave a near linear speed up.

2) In the case of LEA, with the increase in the number of edges locked, the quality of the solution comes down considerably. This is explained by the increase in the number of collisions with more edges locked. When a collision occurs, that part of the search space is ignored by the node. Hence the probability of searching some part of the solution space is reduced.

3) The MLEA performs better compared to the pure locking edge algorithm from the view point of the quality of
solution. This is mainly due to the creation of overlapping search spaces and allocating those search spaces to more nodes on the network. This makes the probability of searching the various solution spaces equally high.

4) The performance of the MLEA with respect to the execution time is comparable with that of the LEA. However, twice the number of nodes are employed in the MLEA compared to the LEA.

5) The TMA is the simplest of the three algorithms and does not involve any problem division overhead. Hence it is better suited to small size problems where the search space is small. When the search space is small and if it is further sub-divided, it may lead to nodes often getting into the search space of neighboring nodes. This may lead to excessive collisions. Hence the TMA is ideal in the case when the search space is small.

In summary, we observe that the TMA is best suited to problem sizes less than $10 \times 10$, the MLEA for problem sizes between $10 \times 10$ and $20 \times 20$ and LEA for sizes above $20 \times 20$.

VI. CONCLUSIONS

Different distributed algorithms have been implemented and their performance measured. The study shows for very large problem sizes locking edges gives better performance. For
medium sized problems, temperature modifier and modified locking edges algorithms give better performance. Tests also show that the distributed algorithms perform better compared to the sequential algorithm with respect to the optimal solution cost and the execution time.

ACKNOWLEDGMENT

The authors thank the anonymous referees for their valuable comments which helped in the revision of this manuscript.

REFERENCES


D. Janaki Ram was born in Andhra Pradesh, India, in 1962. He received the B.Tech. degree in mechanical engineering from College of Engineering, Kakinada, India, in 1983, the M.Tech. degree in computer science in 1985 from the Indian Institute of Technology, Delhi, and his Ph.D. degree from I.I.T., Delhi, in 1990. He is currently an Assistant Professor in the Department of Computer Science and Engineering, I.I.T., Madras. His research interests include distributed computing, distributed databases, object based systems, and CAD/CAM.

K. Krishna was born in Andhra Pradesh, India, in 1970. He received the B.Tech. degree in mechanical engineering in 1991 from S.V. University, Tirupati, and the M.Tech. degree in computer science and engineering from I.I.T., Madras, in 1993. He is currently a Software Engineer at Nucleus Software Group, Madras, India. His areas of interest include distributed computing, networking, and object oriented programming.

K. Ganesan was born in Hyderabad, India, in 1968. He received the B.E. degree in computer science and engineering from the University Visvesvaraya College of Engineering, Bangalore, India, in 1990, and the M.S. degree in computer science and engineering from the Indian Institute of Technology, Madras, in 1993. He has since been working as a Senior Engineer, R&D, for Wipro Infotech Ltd., Bangalore. His research interests include distributed systems, distributed AI, and networking.