Design, implementation and performance analysis of the ant colony optimization algorithm for routing in ad hoc network

by

Mohammad Towhidul Islam

A dissertation submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science
Faculty of Graduate Studies
University of Manitoba

Copyright © 2004 by Mohammad Towhidul Islam

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES
*****
COPYRIGHT PERMISSION

DESIGN, IMPLEMENTATION AND PERFORMANCE ANALYSIS OF THE ANT COLONY OPTIMIZATION ALGORITHM FOR ROUTING IN AD HOC NETWORK

BY

Mohammad Towhidul Islam

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of

Manitoba in partial fulfillment of the requirement of the degree

Of

MASTER OF SCIENCE

Mohammad Towhidul Islam © 2004

Permission has been granted to the Library of the University of Manitoba to lend or sell copies of this thesis/practicum, to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film, and to University Microfilms Inc. to publish an abstract of this thesis/practicum.

This reproduction or copy of this thesis has been made available by authority of the copyright owner solely for the purpose of private study and research, and may only be reproduced and copied as permitted by copyright laws or with express written authorization from the copyright owner.

Abstract

A large class of application where the pattern of data distributions is non-uniform and sparse, is generally classified as *irregular problems*. From the parallel computing perspective, routing in ad hoc network is an irregular problem because, during run-time, the network changes dynamically and exhibits both chaotic load balancing and unpredictable communication behavior among the nodes. The algorithms for these applications are usually asynchronous. One solution technique to respond to these challenges is a swarm based approach called ant colony optimization (ACO), a meta-heuristic which is an inherently parallelizable search technique. One of the many interesting features of ACO is the ability to solve problems that are not static but are spatially distributed and changing over time. In this thesis, a parallel algorithm for routing in ad hoc network is designed and developed. The algorithm is implemented on a distributed memory multicomputer using MPI and on a shared memory multi-processor using OpenMP. The parallel implementation of the proposed algorithm on the distributed environment obtained relative speedup of little more than 7 using 10 processors. In shared memory multiprocessors, extensive experiments are carried out on the several load balancing techniques available in OpenMP. The results gained in OpenMP outperformed the results obtained for the proposed ACO algorithm in MPI regarding the total execution time.

Acknowledgements

I would like to express my profound gratitude to ALLAH, the most merciful, compassionate, who has created me and made me complete this work successfully.

I would like to pay heartily thanks to my supervisor Dr. Parimala Thulasiraman, without whom this thesis would not have been possible. She gave me the opportunity and encouraged me to work in a new research area and at the same time, continued her enthusiastic support throughout the research. In addition, I would like to remember the critical comments and useful suggestion of Dr. Ruppa Thulasiram on my work and thesis which helped me to furnish my thesis. I am thankful to Dr. Peter Graham for his useful comments on my thesis proposal. Special thanks to Dr. Michel Toulouse and Dr. Robert D. McLeod for being in my thesis committee.

I am also thankful to the members of our parallel processing group whose weekly discussion in various aspects stimulates my work.

I never repay the debt of my parents whose endless support, encouragement, and patience make me complete the thesis. Thanks to my brothers and sister-in-laws. And to my nieces, Afifa and Maryum, whose remembering just makes me know what I want to do.

I would like to acknowledge the financial support form Graduate Studies, Faculty of Science, Department of Computer Science and as well as the University of Manitoba Student Union for the University of the Manitoba Graduate Fellowship, the Science Graduate Student Scholarship, the Computer Science Graduate Fellowship and the Academic Excellence Scholarship respectively.

Contents

1 Introduction **1**

2 Parallel Computing Environments **6**

2.1 Flynn's Taxonomy 7

2.1.1 Single Instruction Multiple Data 7

2.1.2 Multiple Instruction Multiple Data 9

2.2 Distributed Memory Multicomputers 10

2.3 Shared Memory Multiprocessor 12

2.4 Multithreading 14

2.4.1 Tera 15

2.4.2 Java 15

2.4.3 EARTH 16

2.4.4 Cilk 16

2.4.5 PThreads 16

2.4.6 OpenMP 17

2.5 Summary 19

3 Ad Hoc Networks **20**

3.1 Routing Protocols 22

3.1.1 Proactive Routing Protocols 22

3.1.2 Reactive Routing Protocols 22

3.2 Summary 24

4 Overview of the Ant Colony Optimization 25

4.1 Foraging behavior of ants 26

4.2 Ant Colony System 27

4.3 Applications of ACO 30

4.3.1 ACO and Communication Networks 32

4.3.2 Routing in ad hoc networks 33

4.4 Parallel aspect of ACO 34

4.5 Summary 36

5 Algorithm 37

5.1 Methodology 37

5.1.1 Assumption 37

5.1.2 Routing table 38

5.1.3 Route discovery 38

5.1.4 Cycle Detection 39

5.1.5 Source update of RT by FANT 40

5.1.6 Route establishment 40

5.1.7 Benefit of source update 41

5.2 Algorithm 42

5.3 Demonstration of Algorithm using an Example 45

5.4 Summary 48

6 Experimental Result 49

6.1 MPI Results 49

6.2 OpenMP Results 54

6.3 Summary 57

7 Conclusion 59

8 Contributions 61

9 Future Work 63

List of Tables

5.1 Dummy routing table for explanation. 45

5.2 Routing table of node 4 before source update. 46

5.3 Routing table of node 4 after source update. 47

5.4 Routing table of node 5 before source update. 47

5.5 Routing table of node 5 after source and destination update update. . 48

6.1 Comparison of OpenMP and MPI. 58

List of Figures

2.1 Logical view of a SIMD Machine [21] 8

2.2 Logical view of a MIMD Machine [21] 9

2.3 Logical view of a distributed memory multicomputer 11

2.4 Logical view of a shared memory multiprocessor 12

2.5 Example of parallel block execution by an OpenMP code 18

3.1 An example of Ad Hoc Networks 20

3.2 An ad hoc network with three nodes. 21

3.3 An ad hoc network with one additional node. 21

4.1 An example with real ants. Adapted from [19]. 26

5.1 An Example of ad hoc network 46

6.1 Performance results with and without source update 50

6.2 Scalability results with varying number of processors and nodes . . . 51

6.3 Number of processors versus percentage of communication time . . . 51

6.4 Number of processors versus percentage of computation time 52

6.5 Number of ants versus Execution time with fixed number of processors and nodes 53

6.6 Speedup Results 53

6.7 Scalability results for varying number of ants 55
6.8 Scheduling policy results 56
6.9 Performance Result with Varying chunk sizes for 8 ants and 400 nodes 56
6.10 Performance results of the scheduling policies when chuck size is unspecified 57

Chapter 1

Introduction

Inspired by observing the behavior of insects, nature inspired algorithms (or swarm intelligence) are becoming popular in many areas of research [6, 49]. Though the insects live in a dynamic, chaotic environment, they cooperate to survive and learn about their surroundings. Experiments and observations have shown that insects such as ants work together as a family by coordinating their activities, and thereby, performing tasks such as finding food more easily and efficiently.

The first ant colony optimization (ACO) meta-heuristic algorithm using swarm based approach was introduced by Colorni et al. [15] and Dorigo et al. [19] who called their algorithm the ant system. The ant system was inspired by theoretical biological studies [5]. For example, when ants leave their nests to search for food, they randomly choose a path and deposit a fragrant chemical substance called a *pheromone*, which helps other ants follow the trail. When more ants traverse a path, the pheromone deposited on that path intensifies, thereby attracting more ants to that path. The scent of pheromone on these paths evaporates after certain period of time. It is obvious that the longest path from a source (nest) to a destination (food source) would have lost its scent much faster than the shorter path. Thus, more ants would

be attracted to the shortest path with the intense scent to find food. These ants indirectly use stigmergic communication to find the best path.

Many applications have been studied in the literature using the ACO technique and are discussed in chapter 4. One of the interesting features of swarm-based approaches is their ability to solve problems that are not static in nature but are spatially distributed and changing over time. ACO has been applied to many combinatorial optimization problems [9, 34]. One area where ACO emerges as a solution technique is in wireless networks [1, 30].

From mobile telephones to home security, wireless technology has become a way of life in the 21st century and has become possible through research activities related to ad hoc networks. Used by military, for example, ad hoc networks are applied where the geographical nature of the system cannot be determined and therefore requires a distributed network topology. A mobile ad hoc network (MANET) consists of mobile wireless nodes that communicate in a distributed fashion without any centralized administration. Due to the node's mobility, it is difficult to determine a network topology that the nodes can utilize at any given time to route the data. The nodes form a network on the fly instantaneously and dynamically when they are needed. Therefore, ad hoc networks are also called "infrastructureless" networks. Ad hoc networks are autonomously formed with heterogeneous devices or nodes from sensors, PDAs or laptops. These nodes range in stability, power capability and processing power. The nodes together co-operate to perform a task such as routing packets. Nodes in a MANET may enter or leave a group as needed.

The ACO technique is quite amenable to ad hoc networks due to similarities in their characteristics. Since the communication links may change dynamically, a node routes packets depending on the link conditions. Similarly, an ant can also exploit the link conditions by altering the amount of pheromone it deposits on a trail. Therefore,

ACO is deemed as an appropriate solution technique for MANETs.

An ad hoc network can be represented as a graph, where vertices represent a set of ad hoc nodes and edges represent the set of communication links connecting the nodes in the graph. From parallel computing perspective, routing in ad hoc network applications is an irregular problem [13] since the network changes dynamically during runtime, exhibits chaotic load balancing among the processors and communicates unpredictably among the nodes during runtime. In recent years, parallel computing has moved towards distributed computing, grid computing, cluster computing, etc. The idea behind this new technology is to utilize spare idle cpu cycles that are available to solve computationally intensive applications efficiently and fast. In this thesis, I hope to answer the question of whether I can extend this idea of utilizing the computing power of mobile devices implemented in an ad hoc network that follows a dynamic topology, by introducing parallel processing techniques for problems such as routing. This idea, I foresee as a definite possibility considering how the technology is advancing (even workstations have two processors). Most mobile devices have Java and advanced features of Java (such as multithreading) installed on them. I therefore see the future of mobile devices moving towards parallel processing (see for example [27]).

The asynchronocity posed by ad hoc networks adds to greater challenges in effective parallel solution because of the need for dynamic creation of nodes and dynamic load balancing. The data dependencies between the nodes in an ad hoc network is sparse. Sparsity of a network is characterized as having only very few edges connected to a node. In other words, if this network is represented as an adjacency matrix, there would be more 0's than 1's where a 1 represents a link between two nodes. This sparsity creates additional difficulties in partitioning and distributing data among the processors. The performance of a parallel algorithm is expected to

be affected significantly by the way the data is distributed among the processors.

In one of the earlier works, conducted by Islam et al. [26] and in its extension [47], the proposed ACO algorithm is parallelized and implemented on a Beowulf cluster running MPI. In this work, it is assumed that the number of ants is equal to the number of processors. Given N, the number of nodes in the network, and P the number of processors, $\frac{N}{P}$ data is distributed to each processor. The results indicate that the sparsity of the data and the asynchronocity posed by the problem due to remote communications affects the performance of the algorithm. About 90% of the execution time is shown to be spent in communication.

In the standard von Neumann model of programming, such as MPI, long latency operations causes the processor to continuously check the buffer for the arrival of the data, or interleave appropriate instructions in the gap of these latencies, that are not dependent on the arrival of the data. However, the processor does have to spend time probing the network or checking the buffer intermittently and the latency problems are not addressed efficiently. One solution to hide latency is multithreading, where a processor switches between threads (a sequential program) during long latency operations. The processors are therefore, always busy as long as there are enough threads in the system. The multithreaded paradigm is very different from the von Neumann model used by MPI. Therefore, traditional parallel algorithms cannot be directly implemented on a multithreaded architecture. The issues (such as thread granularity, thread partitioning etc.) that need to be addressed in designing a multithreaded algorithm are quite challenging.

To determine the effectiveness of multithreading, in [25] (conducted by Islam et al.), the parallel ACO algorithm has been implemented on a shared memory machine using OpenMP [14]. The OpenMP is chosen as the target parallel programming tool for implementation because: (i) in the recent years, it has established itself as

the standard parallel programming tool for shared memory machines; (ii) it provides multithreading; (iii) it allows experimentation with both coarse-grained and fine-grained parallelism (routing in an ad hoc network is a fine-grained application; that is, the number of instructions for the computation is very few.) and (iv) it provides load balancing facilities for experimentation. OpenMP provides three different load balancing strategies: *static, dynamic and guided.* Each one of these strategies has unique feature that suits best for a given application. Various experiments have been conducted with these load balancing strategies. The experiments indicate that the multithreaded algorithm produces better performance results than the parallel algorithm.

The focus of this research, is to design and develop a parallel ACO algorithm for routing in ad hoc network applications, and implement it on a distributed memory architecture using MPI and on a shared memory multiprocessor using OpenMP. In the literature, parallelization has been considered for many static applications using ACO [10, 17, 35, 40, 44]. However, to my knowledge this is the first work in parallelizing ACO for MANETs, a dynamic application.

The rest of the thesis is organized as follows. In the next chapter, the parallel computing environments in general terms is discussed. The background of ad hoc networks and ACO are briefly described in chapters 3 and 4. In chapter 5, the proposed ACO algorithm is presented. The implementation details and experimentation are described in chapter 6. Chapter 7 concludes. Chapter 8 presents the resulting contributions from the thesis. Finally, in chapter 9, the possibility of extension of this work in the future is briefly discussed.

Chapter 2

Parallel Computing Environments

Parallel computing or *parallel processing* is a technique trying to provide the solution of a single problem using more than one processing element (processor or CPU). Given a large task that is very computationally intensive for a sequential computer, the idea in parallel processing is to solve the problem fast and achieving maximum speedup by utilizing multiple computers efficiently.

A problem can be solved by either performing a *domain decomposition* or *functional decomposition*. In domain decomposition, the data is partitioned and distributed to the processors; while in functional decomposition, the task is subdivided and the individual subtasks are given to the processors. In both these decompositions, the type of parallel architectures, parallel algorithm design, parallel compilers and parallel languages influence the speedup of an application.

In this chapter we will briefly discuss the taxonomy of computer architectures [21] and the standard parallel languages.

2.1 Flynn's Taxonomy

Many large commercial parallel machines are based on Flynn's taxonomy of computer architecture [21]. Flynn classified the parallel computer into four main categories according to the number of instructions and data streams per cycle: Single Instruction Single Data (SISD), Single Instruction Multiple Data (SIMD), Multiple Instruction Multiple Data (MISD) and Multiple Instruction Multiple Data (MIMD).

SISD is the classical von Neumann machine or the sequential computer. This serial machine shows the least concurrency among the aforementioned computer models. Pipelining technology helps to achieve parallelism for SISD through concurrent execution of different processing phases.

MISD is logically a pipeline of functional units that operates on a single piece of data. However, it is very difficult to find applications for MISD machines [20].

SIMD and MIMD models have gained popularity since many applications fall into these architectures. These models are elaborated below.

2.1.1 Single Instruction Multiple Data

Many early parallel computers use a topology where a single instruction operates on different sets of data. This topology is known as Single Instruction Multiple Data (SIMD) model. An SIMD machine is comprised of an array of processing elements (PE) which are controlled by a single hardware unit (Figure 2.1). The controlling unit distributes one instruction at a time to each of the PEs thereby maintaining synchronicity among them. That is, each PE performs the same computation on different data sets. Such models are called as *synchronous* programming models. The connection machine [32] and Massively Parallel Processor [4] are SIMD architectures.

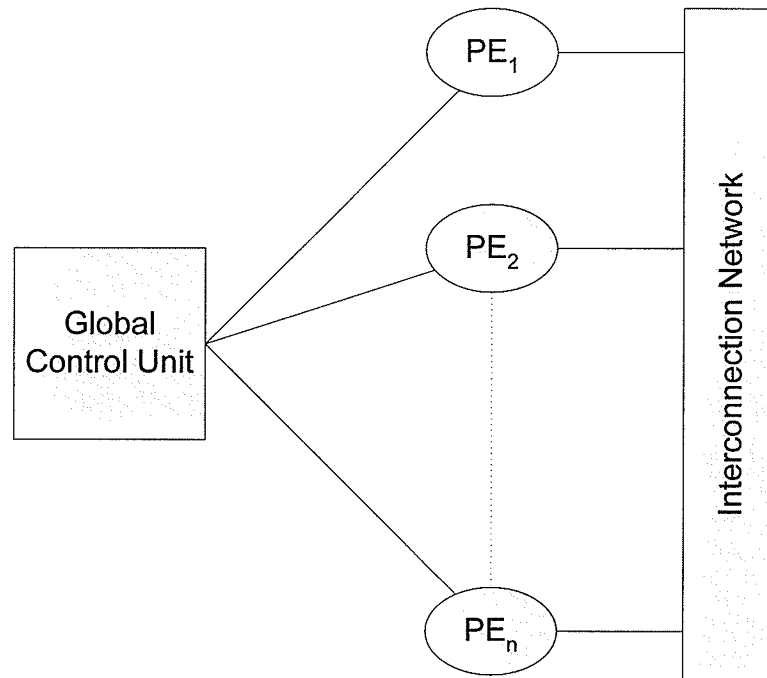

Figure 2.1: Logical view of a SIMD Machine [21]

This type of architecture is useful for vector data manipulation. The optimal performance can be achieved from these machines when the vector size is a multiple of the number of processors. For example, if the vector size is 512 and there are 8 processors then $512/8 = 64$ chunk of data is distributed among the processors. In this case all the PEs are busy executing the computations on their local data set. However, if the vector size is 513, then the first PE has one data more than the other seven processors. The seven processors will be idle while the first PE is busy executing the instruction on its additional data. Since usually in a SIMD architecture such as MPP, the number of processors is large, it is detrimental to obtain maximum effciency from these machines.

2.1.2 Multiple Instruction Multiple Data

In MIMD architecture, autonomous processors execute different programs simultaneously. The processors communicate via an interconnection network. Communication becomes a crucial issue in such machines. These are known as *asynchronous* programming models since they do not require a global clock synchronization as in SIMD machines. Figure 2.2 shows the typical architecture of a MIMD model.

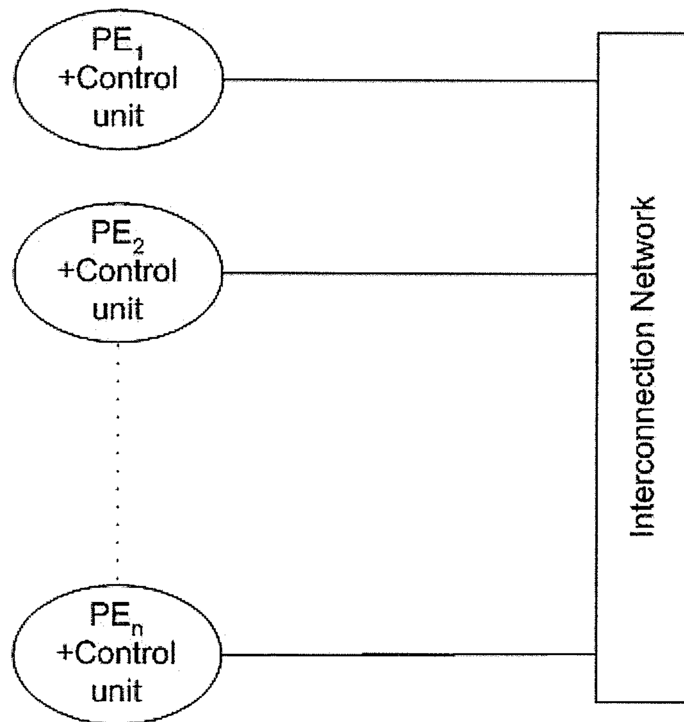

Figure 2.2: Logical view of a MIMD Machine [21]

MIMD machines are also called as *Single Program Multiple Data* (SPMD) models because it is very common to have the same program execute on every processor [29].

The advantage of SPMD over SIMD model is that it can use maximum instruction level parallelism. Moreover, SPMD machines can be built using general purpose processors while SIMD machines require special processors. However, SPMD models mostly suffer from communication overhead among the processors.

The MIMD machines are broadly divided into two categories according to their memory organizations: *distributed memory multicomputers* and *shared memory multiprocessors.*

2.2 Distributed Memory Multicomputers

This architecture consists of several processing nodes (computers) where each node has its own private local memory (Figure 2.3) that is accessible by the local processor only. Since there is no centralized memory, the PEs communicate with each other through an interconnection network using message passing. The topology of the network is very important to the overall evaluation of the performance of the algorithm since the time of the arrival of the messages greatly depends on the interconnection network. The topology of the interconnection network also affects the scalability of the distributed machines. Therefore, many interconnection network topologies exist such as the hypercube, bus, tree and torus [50].

Note that the arrival of messages between nodes is asynchronous adding greater difficulty in designing and implementing parallel algorithms on such an architecture. The sending and receiving of messages therefore has additional overhead in the performance of the algorithm.

Cosmic Cube [43], nCUBE 2 [16], iPSC [2] are the examples of MIMD machines.

Message Passing

Message passing paradigm is an alternative to shared memory programming. The message passing libraries could provide either *blocking* or *non-blocking* communication. In blocking message passing communication, the sender cannot proceed with any other instructions until the data is completely received at the receiving end. In

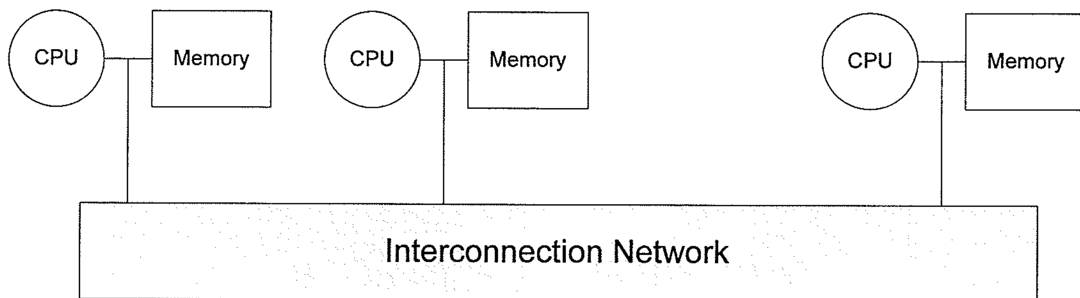

Figure 2.3: Logical view of a distributed memory multicomputer

contrast, in non-blocking communication, the sender places the data to be transferred in a buffer and continues with its other instructions. The sender does not worry about when the message is transferred. The sender can check if the message has been received by the receiver using primitives provided in the message passing libraries. Therefore non-blocking communication provides overlapping of computation with communication.

Message Passing Interface (MPI) and Parallel Virtual Machine (PVM) are two message passing libraries for distributed memory machines. PVM has been designed to form a single parallel computer by connecting several heterogeneous computers together. It is very feasible for heterogeneous machines. PVM provides distributed operating system through its virtual machine by running a deaemon on all the computers.

MPI on the other hand, is now a standard for writing parallel programs using message passing interface. It does not provide a virtual machine like PVM. An object in MPI uses interfaces to communicate with other objects or other resource management system. Portability, modularity and free availability make it possible to use in a wide variety of systems.

In this thesis, we have chosen MPI since it is considered as a standard [22]. More-

over, the experiments are conducted on homogeneous machines such as Beowulf cluster where MPI is more efficient.

2.3 Shared Memory Multiprocessor

In shared memory multiprocessors, the processors have access to the common global memory [21]. Figure 2.4 shows the logical view of this architecture. IBM SP2 and Cray are examples of such machines. There are three models of shared memory multiprocessors and they differ in the organization of the memory and peripheral devices as either shared or distributed. They are *Uniform Memory Access* (UMA), *Non-Uniform Memory Access* (NUMA) and *Cache Only Memory Access* (COMA) models.

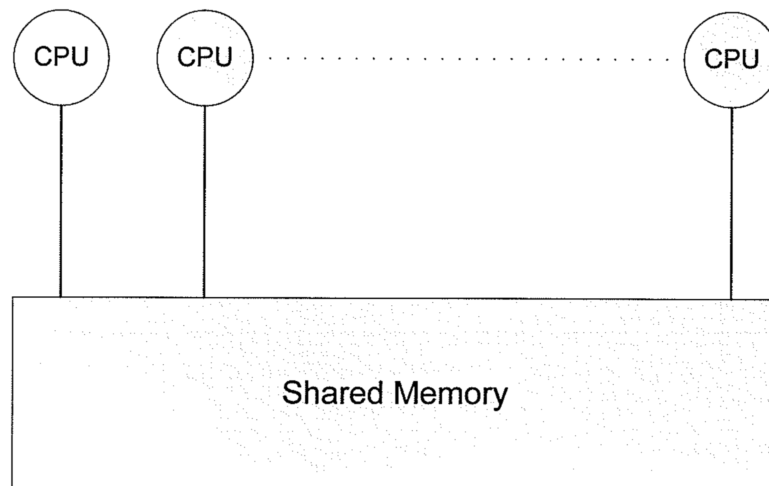

Figure 2.4: Logical view of a shared memory multiprocessor

In UMA model, all the processors have equal access to the physical memories. The physical memories is uniformly shared by all the processors. These machines are also called as *Symmetric Multiprocessors* (SMP). Programming in this model is easy

but the drawback is that it does not scale well to many processors. Locality is of less concern in these SMPs.

NUMA machines are also called as *distributed shared memory machines.* In this model, though each processor has its own local memory, the processors can access every other processors memory. Since some processors memories are closer to others and depending on the location of the processors, the access time varies. The shared address space together forms a global address space shared by all processors. SGI Origin 2000 and Sun Ultra HPC Server are examples of NUMA machines.

COMA machines are a variant of NUMA machines. In this model, the distributed shared memory are the caches. All the caches of the processors together form a global address space.

One of the drawbacks of shared memory machines to distributed memory machines is that since there is one global memory, the resources are shared by the processors. Locks and semaphore constructs are necessary to properly synchronize and co-ordinate the tasks between the processors. When many processors access a particular memory location, *memory contention* arises. Programming, though is conceptually easier in this model, the performance of the algorithm may be limited due to the sharing of the resources.

Each processor has a cache memory which is faster to access than the shared memory. It is possible that a shared data may exist in the cache memories of different processors at a given time. The datum, therefore, in the shared memory and the local caches is invalidated. Cache coherency protocols must be maintained. Cache coherence mechanisms ensure that all processors work with consistent data and can be implemented either in hardware or can be handled by software. For example, snoopy protocol is used in bus based shared memory machines for cache coherency [38].

2.4 Multithreading

In *massively parallel processors*, there are two types of latencies that asynchronicity triggers: *communication* and *synchronization latencies* [3]. Communication latency arises when a processor requires a particular data from another processor and has to remotely access this data via message passing. The processor idles until the data is received. To determine if the data is ready at the receiving end, the processors have to synchronize. In such situations, busy waiting may result. In both these cases, the processor idles.

There are many ways to overcome these latencies: hide, tolerate or reduce. The general technique is multithreading which tries to hide the latencies such as the non-blocking communication protocol in MPI. In multithreading, the main idea is to overlap computations with communications. The processors context switches between threads, where a thread is a sequence of instructions.

In non-blocking communication in MPI [22], a processor (sender) may send a message to another processor (receiver) requesting for information. During this process, the sender, in order not to stay idle, executes the next set of instructions following the send operation, if there is any. In this case, the program must have instructions that it can overlap during the communication phase. Simultaneously, the sender checks the buffer periodically using the primitives provided in the message passing library to determine if the message it has requested has arrived in its buffer. Though there is some possibility of computation and communication overlapping, the periodical checking for the message takes a significant amount of time. This is an additional overhead incurred in MPI programming which is not visible to the user but degrades the performance of the algorithm.

In the general sense of multithreading, the algorithm is divided into many threads,

where each thread is a sequence of instructions. A thread could be fine-grained or coarse-grained. Fine-grained threads contain very few instructions, maybe ten instructions per thread. Coarse-grained threads are larger threads. The program running in the SPMD model can be categorized as coarse-grained threads.

Determining the thread granularity (fine or coarse) is very difficult in a multi-threading environment. Many issues (such as thread boundary, thread size, number of threads) need to be considered while designing algorithms for much paradigms.

Tera MTA multithreaded architecture [8], Cilk [45], Solaris threads and Pthreads [36], Java [37] and OpenMP [14] are all multithreaded machines, libraries or languages that exist in the literature.

2.4.1 Tera

Tera is a multithreaded architecture [8] that provides super pipelining in its processor-network-memory operation for high performance computing. Tera can handle 128 contexts per processor and it performs a context switch per clock cycle with numerous threads or context. Tera supports latency tolerance rather than latency reduction. Tera is used in research labs such as the San Diego supercomputing center where the machine is installed and in Nasa Ames [33].

2.4.2 Java

Java is an interpreter based language which supports both single processor and multiple processors. Portability of Java code is a major advantage. However, Java is not a good choice for fine-grained parallelism as context switching between processors require significant amount of time [37].

2.4.3 EARTH

Efficient Architecture for Running THreads (EARTH) [24] is a multithreaded dataflow architecture that does not follow the traditional von Neumann model of computing in traditional parallel computers or the Tera multithreaded architecture. EARTH threads are very versatile and support both fine-grained and coarse-grained computing. However, the language used by this architecture called Threaded-C is not a user-friendly language. In EARTH, threads are scheduled depending on control and data dependencies, which circumvent synchronization and communication latencies quite significantly. Though there is a simulator version of the EARTH and is possible to access it remotely, remote access is always very difficult.

2.4.4 Cilk

Cilk is a multithreaded language based on a ANSI-C programming language. Cilk is suitable for dynamic and highly asynchronous applications which follow a recursive programming model. Cilk employs a runtime scheduler to accurately estimate and monitor the performance of a program. The load balancing and communication protocols are also managed by Cilk runtime system [45].

2.4.5 PThreads

POSIX Threads (Pthreads) are operating system threads that are primarily developed for Linux and Unix environment to write programs that require concurrency. PThreads is a library based multithreaded approach for coarse-grained parallelism [36]. Pthreads are preemptive threads and the context switching time between threads is significant.

2.4.6 OpenMP

OpenMP is an application programming interface (API) for parallel programming which comprises of a set of directives and libraries. OpenMP is designed for shared and distributed shared memory multiprocessors. OpenMP can convert a existing sequential code into parallel code using three of its features: directives, library routines, and environment variables. It is easy to include OpenMP API in conventional C/C++ or Fortran language [14].

The directive approach of OpenMP makes it portable API in the multithreaded paradigm. It is easy to write and compile a parallel code using directives. The directives are treated at compile time. Therefore, the code written in OpenMP can be optimized at compile time. On a non-OpenMP environment, the directives are considered as comments and discarded. The directives, therefore, facilitate porting the OpenMP code on a nonOpenMP environment. Hence, it is only necessary to recompile the whole code for porting into a new system.

OpenMP exploits parallelism using control structures. OpenMP uses fork/join model for multiple threads. When a *parallel* directive encloses a block, the block could be executed by multiple threads concurrently. There are two types of parallel constructs: *loop level parallelism* and *sections.* In *loop level parallelism* threads could be divided as fine-grained threads. In this method, a loop (equivalent *for*) may be parallelized by allowing each iteration of the loop to be executed by a thread. Note that each thread executes the same code on different data sets following a SPMD model. This allows efficient work sharing among the threads. The scheduler distributes the workload dynamically among the processors. On the other hand, threads could execute different pieces of code concurrently. OpenMP uses *section construct* for work sharing. Each section contains a block of code and each thread

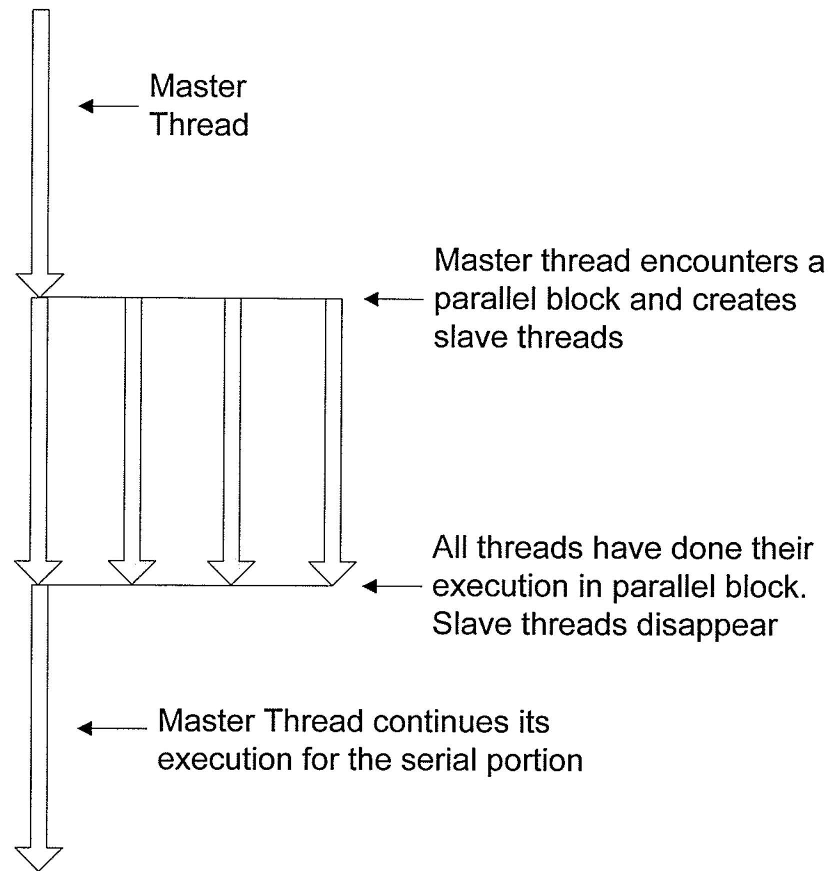

Figure 2.5: Example of parallel block execution by an OpenMP code

executes a block enclosed in a section; this is more coarse-grained.

The OpenMP program starts with a single thread. This thread is known as a master thread. When a master thread encounters a parallel block, it creates slave threads according to the request of the block. Thus, the master thread together with the slave threads execute the parallel block. When a thread completes its work, it waits for other threads to finish. At the end of the block execution when all threads have finished their computations, the slave threads disappear. The master thread continues its execution as a serial program. Figure 2.5 shows the execution of the parallel block in an OpenMP environment [14].

2.5 Summary

This chapter summarizes the parallel computing environments. In this thesis, MPI and OpenMP are employed for efficient parallel implementation of the proposed parallel ACO algorithm. MPI runs on a network of workstations and 10 node Beowulf cluster at University of Manitoba, Computer Science department. OpenMP runs on an 8 node shared memory machine also available in the Computer Science department.

Chapter 3

Ad Hoc Networks

An ad hoc network is a collection of mobile nodes that routes packets without any central administration or standard support services that are available on wired networks. Laptop computers and personal digital assistants that have independent communication capability are examples of nodes in an ad hoc network. Figure 3.1 shows a simple ad hoc network comprised of three devices.

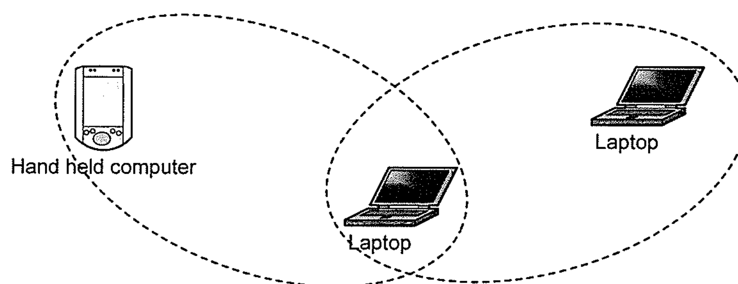

Figure 3.1: An example of Ad Hoc Networks

The mobile nodes in an ad hoc network are operated by low powered batteries which limit their transmission range to the nodes that are closest to them. For example, consider Figure 3.2 with three wireless mobile hosts. The circles around the nodes indicate the transmission range of these nodes. Nodes 1 and 2 are reachable

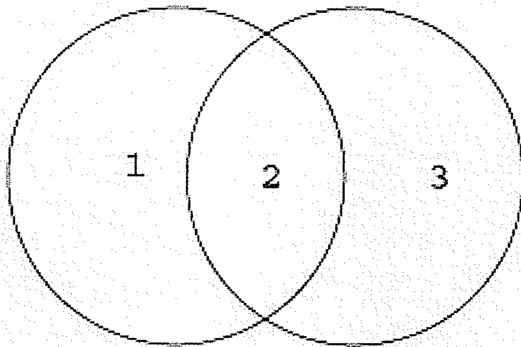

Figure 3.2: An ad hoc network with three nodes.

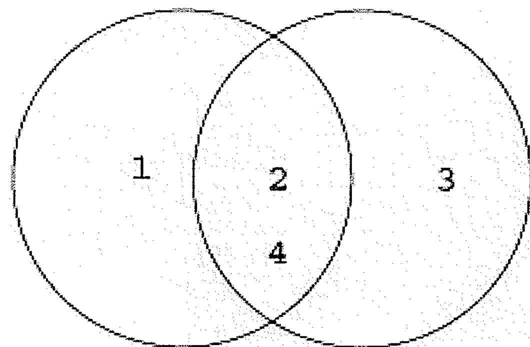

Figure 3.3: An ad hoc network with one additional node.

from one another, as are nodes 2 and 3. However, node 3 in this case is not in the transmission range of node 1. If node 1 and 3 wish to communicate (route packets), they need to get the aid of the intermediate node, node 2, which is within the transmission range of both node 1 and node 3. A node, therefore, acts as both a host and a router transmitting data to other mobile nodes that may not be within transmission range of each other. Suppose another node is added to the above network (see Figure 3.3). The situation changes considerably, since there are now multiple paths between nodes 1 and 3. Packets can be routed along paths 1-2-3 or 1-4-3 or 1-2-4-3. The situation becomes more complicated when more nodes are involved in the network. An ad hoc routing protocol must, therefore, determine the "best" path to route the packets between the nodes. The ad hoc network is therefore characterized as a dynamic topology since the mobile nodes move frequently. This characteristic in an ad hoc network demands routing protocols that dynamically discover routes [31].

3.1 Routing Protocols

The ad hoc routing protocols may be classified as proactive routing protocols or reactive routing protocols. In proactive protocols, all nodes keep one or more tables of up-to-date information about all other nodes in the network. When a node joins or leaves the network, update messages are relayed to all the nodes. Reactive routing protocols create routing tables when they are required. Not all updates are preserved at every node. In the following sections, a brief explanation of the protocols is given.

3.1.1 Proactive Routing Protocols

In this approach, each node in the network tries to keep an updated copy of information about all the nodes in the network. Routing protocols differ in the way they maintain the routing information and methods by which they build the routing table. Several routing protocols have been developed following the proactive methods. Destination Sequence Distance Vector (DSDV) [39] is one such protocol. This protocol is based on the Bellman-Ford [48] routing algorithm, where each node V contains information about all the nodes in the network and their distances from V in terms of the number of hops. To make the network consistent, each node sends its status to every other node, which generates a lot of control packets as well as creating inefficient traffic in the network.

3.1.2 Reactive Routing Protocols

When a node (the source) wishes to send data to a particular node (the destination), the source finds a route to the destination by initiating a route discovery phase. Once the route is established, only the nodes in the route participate in the route maintenance phase. There are two main protocols: Ad Hoc On-Demand Distance

Vector Routing (AODV) [41] and Dynamic Source Routing (DSR) [28] that apply reactive routing approach. AODV is an improved version of DSDV. The source node broadcasts a route request (RREQ) packet to its neighbors and the RREQ packet is sent to the destination through intermediate nodes. When the RREQ packet reaches the destination or an intermediate node, which has the latest information about the destination, a route reply (RREP) packet is sent back to the source. The RREP packet follows the reverse path of the RREQ packet and sets up the routing tables of the nodes along the path to forward data from the source to the destination. Since the RREP packet follows the same route as the RREQ packet, the links in AODV networks must be bidirectional. If a node along the path is moved or a link is removed, the neighboring nodes of the recently removed node send the broken link information to the source. The source then re-initiates the path. For this purpose, every node keeps track of its neighbors along a particular route [41].

In DSR, each mobile node contains a *route cache* to keep the routing information about other nodes in the network. When one node V tries to send data to another node U, V first looks at its route cache. Node V sends data directly to the destination if routing information is found in its route cache. Otherwise, node V sends a *route request* packet towards the destination. This packet also contains the *route record* and keeps information about the nodes along the path.

A route reply packet is sent by an intermediate node or by the destination node. If any intermediate node has information about a route to the destination, it appends this information into the route record and sends it back to the source. If there is no such intermediate node on the path to provide routing information, the route request packet reaches the destination. The destination extracts the route record information from the route request packet and appends this information to the route reply packet. Then the destination sends this route reply packet to the source node and the path

is established.

The route is maintained using *route error packets.* When one node experiences a transmission error through a link, it yields a route error packet and notifies all other nodes using this link. The nodes in the network update their route caches accordingly [28].

3.2 Summary

In this chapter, the DSDV protocol is primarily discussed since it closely resembles the traditional protocol for wired networks. However, DSDV does not show good performance in highly mobile environment. Therefore, the originators of DSDV proposed a reactive version of DSDV called AODV. The multicasting capability of DSDV enhance the performance of DSDV when one node communicates with several nodes. The route discovery phase of DSDV resembles the route discovery phase of DSR. Both use a route request packet from source to destination [31]. In the proposed parallel ACO algorithm discussed in the next chapter, the source node sends active agents (alias ant) to the destination node and the destination node returns another agent to source for route establishment which is similar to the route reply packet in DSDV and DSR.

Chapter 4

Overview of the Ant Colony Optimization

Swarm intelligence [6] is a research area inspired by observing the behavior of insects. Scientists have been looking at the collective behavior of insects such as ants, honeybees, wasp, etc. for solving several NP-Hard optimization problems. By observing the behavior of ants, a new metaheuristic has been developed called the Ant Colony Optimization (ACO). A metaheuristic can be defined as a high-level algorithmic approach that can guide many heuristic problems. The first ACO (a subset of swarm intelligence) was introduced by Dorigo [19]. The ACO algorithm is based on the techniques used by ants that collectively participate and collaborate in activities such as finding food or building a nest. Ants are very intelligent and self-organized insects. This observation has led to solving many static and dynamic problems.

In this chapter, the basic idea behind ACO is discussed in section 4.1. Section 4.2 gives a high level description of the ACO algorithm. In section 4.3, the applications of ACO is described. The parallel aspects of ACO is explain in section 4.4.

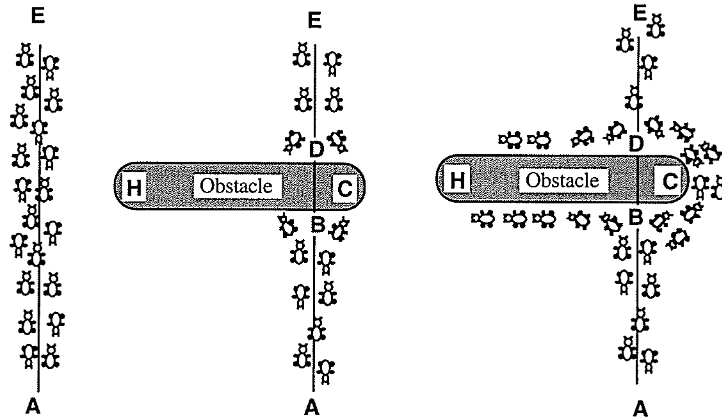

Figure 4.1: An example with real ants. Adapted from [19].

4.1 Foraging behavior of ants

The basic idea of ACO comes from the foraging behavior observed by the colony of ants. When ants leave their nest in search of food, they deposit a chemical substance called *pheromone* on their path. The scent of pheromone lures more ants to follow the same path. When more ants traverse a link, the amount of pheromone deposited on that link gets intense, thereby attracting more ants to that trail to travel from the nest to the destination and vice versa. The scent of the pheromone on these links evaporates after certain amount of time. The longest paths from source to destination lose their scent much faster than the shorter paths. Thus, more ants choose the shortest path to find food. This indirect communication among ants is known as *stigmergy*.

We can explain the aforementioned phenomena using Figure 4.1. Assume the nest is located at A and the food source is at E. The ants travel back-and-forth between A and E. Assume, suddenly an obstacle is placed on their path. At position B, an ant has to decide which way to traverse. An ant could either take BHD or BCD. The same scenario will occur at position D for the ants coming from E. An ant at position

B has equal probability to turn left (taking BHD) or turn right (taking BCD). Let us assume that half of the ants at point B take left turn while the other half of the ants take right turn. The ants taking BCD reach D quicker than the other ants traversing BHD since the path is shorter. Also, note that the pheromone concentration level evaporates in course of time. Since the BHD path is longer than the BCD path, the chances of losing the scent in this path is much faster than that of path BCD. Hence the ants at point D returning from E take DCB. This phenomena eventually increases the pheromone concentration on the shortest path thereby establishing a route between A and E via BCD.

4.2 Ant Colony System

Dorigo, Maniezzo and Colorni [18] first applied the foraging behavior of ants to the solution of the the path optimization problem, the travelling salesperson problem (TSP). They chose TSP because it is an NP-hard problem and the ant system metaphor can be easily adapted to this problem [6]. The idea is to distribute a set of ants is distributed on several cities. Each ant starts traversing from its current city, visits each city exactly once, and returns to its original city completing a tour. On their journey, the ants deposit the pheromone along the edges connecting the cities. After one complete tour, the pheromone concentration is adjusted along the edges and the ants start their tour again. In this process, the edges that are not part of the solution will have less pheromone and eventually evaporate.

Dorigo et al. [19] first developed Ant System (AS) for TSP. However, the result for TSP from AS was not optimistic. Since AS performed well for small networks, the authors later proposed a modification of AS called Ant Colony System (ACS) [18] or ant colony optimization (ACO) algorithm.

The ACO algorithm works as follows. Given m ants and i_{max} iterations, the algorithm alternates between two basic phases [6]:

- Construction Phase: Each of the m ants develop and construct m solutions to the problem under consideration. The ants perform this construction in parallel.
- Pheromone Update Phase: The pheromone concentration level on the edges is modified.

The ACS provides three rules: transition rule, global update rule and local update rule.

Transition rule: The construction phase is probabilistic. Each edge (r, s) connecting vertices r and s is associated with a length $d(r, s)$. The *visibility* $\eta(r, u)$ is defined by $1/d(r, s)$. Visibility can be regarded as heuristic desirability, that is, choosing vertex s from vertex r. This selection strictly depends on local information and is used to allow an ant to select the next node on its journey to the destination. $\tau(r, s)$ is the pheromone concentration on the edge between two vertices r and s. τ is not static since it is updated during the problem solution. It reflects the past history of the edge and the ants desirability to choose that edge. More than one ant may have traversed an edge and that is reflected in τ.

The probability of adding a new element to the solution space constructed by an ant is a function of the element's heuristic desirability η and the pheromone trail τ.

An ant k at vertex r chooses to move to a vertex s using the following transition rule:

$$s = \begin{cases} arg\ max_{u \in J_k(r)}[\tau(r, u)] \cdot [\eta(r, u)]^{\beta} & \text{if } q \leq q_0 \\ S & \text{if } q > q_0 \end{cases} \tag{4.1}$$

Here q is a random variable uniformly distributed over [0,1]. q_0 is a tunable parameter where $0 \leq q_0 \leq 1$. $J_k(r,s)$ is the set of unvisited nodes by this ant. β is a user defined parameter that is adjustable. It controls the relative weight of η. S is selected randomly from the set of unvisited nodes, $J_k(r)$, using the following probability:

$$p_k(r,S) = \frac{[\tau(r,s)] \cdot [\eta(r,s)]^\beta}{\sum_{u \in J_k(r)} [\tau(r,s)] \cdot [\eta(r,s)]^\beta} \tag{4.2}$$

When $q \leq q_0$, the transition rule follows the exploitation method. It uses the knowledge of the distances and the pheromone trail to exploit an edge. For, $q > q_0$, the transition rule follows the exploration method of trying to choose an edge that is unvisited.

Global pheromone updating rule: The ACS gives emphasis on the probable best solution. After completing the tours by all ants, the edges that belong to the best tour are rewarded. The pheromone level of this best tour is increased by the following rule:

$$\tau(r,s) = (1-\rho) \cdot \tau(r,s) + \rho \cdot \Delta\tau(r,s), \ where\ \Delta\tau(r,s) = \frac{1}{best\ tour\ length} \tag{4.3}$$

This update encourages the ants to follow the optimistic path to find the optimal solution. Local updates are also performed to find other solutions.

Local pheromone updating rule: When an ant k at vertex r selects the next vertex $s \in J_k(r)$, the pheromone concentration of edge (r,s) is updated by the following equation:

$$\tau(r,s) = (1-\rho) \cdot \tau(r,s) + \rho \cdot \tau_0 \tag{4.4}$$

τ_0 is the initial pheromone level and ρ is the decay parameter. The local updating rule actually decreases the pheromone concentration level on an edge. This is because,

it was found experimentally by Dorigo et. al. [18] that by allowing visited edges to be less attractive, the ants may choose other paths and eventually find an optimal solution. The argument behind this is that, when an ant eventually finds a path it may not necessarily be the shortest path.

A high level description of the ACO algorithm is given below [6]:

- Initialization: for all edges (i, j) set $\tau(i, j) = \tau_0$.
- Given m ants, place an ant on each of the randomly chosen vertices.
- For i_{max} iterations perform the following:
  - Each of the m ants construct a solution to the problem by using the transition rule.
  - Each ant k computes the cost of the solution obtained above.
  - If an improved solution is found then update best solution found by using the global update rule.
  - For all edges (i, j) use the local update rule.
- Print the best solution.

4.3 Applications of ACO

ACO was initially applied to various well-known combinatorial optimization problems. They found that the ACO algorithm for these static problems gave better results than the other existing solutions for the problems under study. This encouraged others to study various communication network problems that are more dynamic.

Bullnheimer et al. [9] proposed an application of the ant system to vehicle routing problem (VRP). Given n vertices, one vertex is the depot and the rest of the vertices

are customer locations. Each edge is associated with a parameter, d_{ij} which represents the distance between two locations. Each vertex, besides the depot vertex, is associated with a demand and service time. There are m vehicles and each vehicle can hold a certain load capacity. The vehicles are allowed to visit each location exactly once without exceeding its total demand capacity and not exceeding its maximum travel distance. Every vehicle must start and end at the depot. The objective of the VRP is therefore, to minimize the cost of delivery by vehicles to customers from the depot subject to the constraints mentioned above. As can be noted, VRP is nothing but the TSP with some limitations. The same technique used in the TSP is applied to this problem to find the solution. The results obtained for VRP using ant system indicate that it performs better than simulated annealing [46].

Maniezzo and Colorni [34] use ACO to solve the quadratic assignment problem (QAP), an NP-Hard problem. In QAP, given n facilities and n locations, each of the facility has to be assigned to a location. The problem can be represented as a graph, where each node in the graph is a location. Each edge connecting locations i and j is associated with a distance parameter d_{ij}, which can be represented as a matrix. We are also given another facility matrix, where a flow, f_{hk} is associated between two facilities h and k. There are $n!$ possible assignments. The cost of each assignment can be computed by multiplying the flow between each pair of facilities by the distance between their locations and summing over all the pairs. The solution to the QAP using ACO is very similar to that used in the TSP. The results outperformed genetic algorithm but not simulated annealing [46].

Another NP-hard problem is the job scheduling problem (JSP) [6]. Given M machines and J jobs, each job j contains an ordered sequence of operations. Each of the operations of a job has to be executed on a machine m during consecutive time periods. The goal is to minimize the total time for the completion of all jobs

in the given time intervals while maintaining the constraint that no two jobs will be processed on the same machine at the same time. Colorni et al. [6] represents the JSP as a directed weighted graph. Each vertex in the graph represents an operation O_i of machine m_i $(i = 1, 2, ..., M)$and job j_k $(k = 1, 2, .., J)$. Using the ACO algorithm, each ant starts adding nodes to its solution space. If a node is already selected, it will not be visited again. A new node is selected as the next node in the solution space, if this node corresponds to the next operation of the same job of the previously selected node. Colorni and Dorigo obtain reasonable performance using ACO for 10 jobs using 15 machines [6].

4.3.1 ACO and Communication Networks

In this section, a brief overview on the related work conducted on some of the problems in communication networks using ACO is described.

Schoonderwoerd et al. [42] proposed the idea of using an adaptive routing algorithm in telephone networks called ant based control (ABC) algorithm. In this algorithm, ants are placed at various locations in the network. The goal of the ants is to maximize performance of the network by adjusting the routing tables located at each of the nodes to adapt to the load changes in the network. The ants route traffic along paths that are less congested thereby achieving better throughput from incoming calls. The algorithm adjusts the parameters according to the demand of the network. Schoonderwoerd et al. noticed that ABC resulted fewer call failures than the other existing algorithms.

Inspired by ABC, Di Caro and Dorigo [12] later proposed another novel algorithm called AntNet for different types of routing protocols such as packet switching or circuit switching in data networks. AntNet uses two homogeneous ant like mobile

agents, *forward ant* and *backward ant*, for exploring the paths from a source to a destination. The source node sends a forward ant towards destination node. When the ant reaches the destination node, it transfers its experiences (pheromone trail, heuristic desirability) to the backward ant about the network. The backward ant, as it is travels back to the source node, updates the routing table at each node. The entry in the routing table contains information about the probability of choosing a path. The update ensures that the best path to the destination achieves the highest probability. The advantage of AntNet is that it keeps additional information such as reliability of a path in the form of estimated trip time (the time required to traverse a path). However, in AntNet, the forward ant does not participate in the modification of the routing table of the visited nodes.

4.3.2 Routing in ad hoc networks

The ACO technique is quite amenable to ad hoc networks due to their similar characteristics. An ant creates a path dynamically just like the routing protocols in MANET. The communication between ants is very minimal, as in MANET. Since the communication links may change dynamically, a node routes packets depending on the link conditions. Similarly, an ant can also exploit the link conditions in the amount of pheromone it deposits on a trail.

The successful implementation of ABC and AntNet lead to the use of artificial ants in a mobile ad hoc network. There already exists two protocols in this area. Câmara and Loureiro [11] describe a novel routing protocol GPSAL (GPS/Ant-Like Routing Algorithm) for MANET. They use mobile software agents modeled as ants to update network information. The ants collect information about the location of the nodes in the network and disseminate this location information to the mobile hosts. However,

their approach ignores the notion of stigmergy and concentration of pheromone on the trail. Güneş et al. [23] extend the idea and consider the stigmergy effect of the pheromone deposited by the ants. This pheromone is used as a medium of indirect communication both in the route discovery and route maintenance phases. However, their system is not scalable for wide area networks and cycle formation between mobile hosts, a common phenomenon in MANET, is not detected. Recently, Arabshahi et al. [1] have considered an energy-conserving routing algorithm for MANET.

4.4 Parallel aspect of ACO

The ACO technique is inherently parallel, but little research has been done in this aspect. The efficiency of parallelization of ACO algorithm depends on the application. The experimental platform is also important to produce efficient results. Simulation modeling, distributed multicomputers or shared memory multiprocessors have been used in the literature, for parallelization of the ACO algorithm.

Bullnheimer et al. [10] first investigated the parallelization of the ant system on the TSP. The problem is studied from a synchronous and a partially asynchronous perspective. In the synchronous algorithm, a processor acts as a master and each slave processor sends a completed tour and the length of the tour to the master processor. Therefore, the frequency and volume of communication is high in the synchronous approach. This synchronization and communication overhead reduces the performance of the algorithm. To circumvent the problem, they consider a partial asynchronous approach [10]. Bullnheimer et al. analytically showed the performance gain of their algorithm. They simulated their algorithm on a discrete event simulator to evaluate the performance gain and speed up of their parallel ACO algorithm on TSP.

Stützle [44] modified the Ant System for parallelization and called it *Max-Min system* for parallelization. In Max-Min system, only one ant can modify the pheromone concentration on the trails after completing an iteration. The level of pheromone should be within a maximum and minimum bound. Thus the algorithm offers more opportunities to explore paths [44]. Stützle experimented the algorithm on a distributed architecture to show the efficiency and speedup of his algorithm. The algorithm was implemented using parallel independent run [44] which is the simplest way to parallelize a program. That is, each processor or each working ant independently finds the solution to the problem. The advantage of this method is that there is no communication overhead among the processors. Stützle showed that the parallel algorithm gets speedup upto P where P is the number of processors.

Randall and Lewis [40] explained various parallel decomposition strategies and specifically applied them to the TSP. They evaluated *Parallel Ants* scheme as a parallelization technique based on ACO. However, the parallel ants are not independent as described by Stützle. They used master/slave approach. The master processor generates the input graph, distributes the ants to different nodes in the network and also updates the pheromone trails. The slave processors perform their operations on their own data. After completing their iteration, they send the pheromone matrix for the trails to the master processor and get the updated pheromone matrix from the master processor. Randall et al. implemented the aforementioned method on the IBM SP2 architecture. They used MPI for communication between the master and slave processors. They showed that a significant amount of speedup can be obtained for larger problem sizes by parallelization of ACO technique. The algorithm, however, is more centralized.

Delisle et al. [17] described another ACO parallelization technique for the industrial scheduling problem and implemented the algorithm on a shared memory

multiprocessor using OpenMP. They used a shared memory machine to eliminate the large communication overhead experienced by their algorithm on a distributed memory machine. They showed that for their application, the ACO technique can achieve good performance gain. Moreover, they experimented their algorithm by increasing and decreasing the number of ants to allow proper sharing of load. This could be easily accomplished on a shared memory machine compared to a distributed memory machine.

4.5 Summary

In this chapter, an overview of the ACO algorithm, sequential and parallel work that has been conducted in this area was discussed. To our knowledge, ACO has not been parallelized for ad hoc networks. The next chapter describes the proposed ACO algorithm for routing in ad hoc networks.

Chapter 5

Algorithm

In this chapter, the proposed algorithm for finding route in ad hoc network is described. A demonstration of the algorithm using an example is given at the end of the algorithm description.

5.1 Methodology

In this algorithm, two homogeneous ants (act like agents), forward ant (FANT) and backward ant (BANT), are used. The FANT discovers the route from a source to a destination while the BANT establishes the final route.

5.1.1 Assumption

The network can be represented as a graph $G = (V, E)$ where V is the number of vertices (nodes) and E is the number of edges (links). We apply the ACO meta-heuristic search algorithm to find the shortest or best path from a given source to the destination. Each link $e = (v_i, v_j)$ is associated with two variables: $\varphi(v_i, v_j)$ represents the pheromone value on each link and $w(v_i, v_j)$ represents the time (the time required

by a packet to traverse the link). The pheromone value gets updated by the ants as they traverse the links. The ants change the concentration of the pheromone value on their journey to the destination and on their way back to the source.

5.1.2 Routing table

Each node contains a routing table (RT). The size of RT is the degree of the node times all the nodes in the network. That is, if we assume the number of nodes in the network is N and degree of node v_i is d_i, then the size of the table is Nd_i. The rows indicate the neighbors of node v_i and the column represents all the nodes in the network. Since the number of nodes in an ad hoc network is small, this routing table is feasible. There are two entries in the routing table for a particular (row, column) pair: a number indicating whether the node has been visited by other ants and the pheromone content.

5.1.3 Route discovery

When a source node S wishes to send data to a destination node D, node S tries to establishe a route to node D by sending a FANT towards the destination node. When a FANT arrives at a node v_i from a source S to travel to a destination D, it considers node v_i's RT to select its path or the next hop neighbor. It considers node v_i's neighbors, v_j, by looking at each row in the routing table and column D. The FANT first selects a node that has not yet been visited by other ants. The purpose of selecting an unvisited node is as follows: assume a FANT randomly chooses a node v_j from v_i to traverse and adjusts the pheromone concentration. Let us assume that another FANT also reaches the same node heading to the same destination node D. Since the pheromone value on the trail considered is higher (due to its recent traversal)

than the other neighboring trails, the FANT may select this node again. However, the FANT does not know if this path leads to the best path. Ants may be following a trail that may lead to a longer path. To avoid this situation, the algorithm *explores* all nodes not yet considered (visited) before relying on the pheromone value. This algorithm can be considered as an *exploration* technique rather than an exploitation mechanism.

If there does not exist an unvisited node, the FANT searches for the next hop node by considering the pheromone concentration. In the RT, the FANT looks at the rows of column D for greater pheromone value. The value of the pheromone that is the largest is regarded as the next hop neighbor. The greater the pheromone concentration for a particular neighbor v_j, the greater is the probability that this node will lead to the best path.

5.1.4 Cycle Detection

Before selecting a node as a next hop, the FANT determines if it has already visited the node before. This is to ensure that the FANT does not travel in a cycle. Each FANT, therefore, holds a list (called $VisitedHop$) of all the nodes that have been visited by the FANT on its current journey to D. The FANT also keeps a stack data structure, which contains all the nodes that may give a promising path to D. The maximum size of the stack is $|V|$. If due to obstruction in the environment, it returns to the same node that has already been visited, indicating a cycle, the FANT immediately backtracks to the previous node from where it came from by using the stack data structure. Also, note that there maybe a situation where there is no path from the current node (a dead end). In this case, the ant uses the stack to backtrack.

5.1.5 Source update of RT by FANT

An ant keeps in its memory the total time (T) it has travelled thus far. If a node v_j is selected as the next hop of node v_i, the FANT moves to the next node v_j and the pheromone update is as follows for entry (v_i,S) in v_j's RT:

$$\varphi(v_i, v_S) = \varphi(v_i, v_S) + \frac{\epsilon}{T(v_S, v_i) + w(v_i, v_j)} \tag{5.1}$$

where ϵ is a runtime parameter provided by the user and $T(v_S, v_i)$ is the time to travel from the source to v_i. On all other nodes in column S the pheromone value is decremented by

$$\varphi(v_l, v_S) = (1 - E)\varphi(v_l, v_S), \forall\, l \neq i \tag{5.2}$$

where E is the evaporation rate of the pheromone. E is a variable parameter also provided by the user. Since we are updating the source column in the RT we call this as *source update* technique. Each FANT also records the total time of the path just traversed as $T(v_S, v_i) + w(v_i, v_j)$.

5.1.6 Route establishment

When the FANT reaches the destination node D, it transfers its memory to a BANT and dies. The BANT follows the same trail as FANT to reach the source node S from the destination node D. It uses the stack to backtrack to the source. On its way back to the source, the BANT again updates the pheromone concentration. However, it updates the *destination* column of RT. For example, the BANT at node v_k travelling backwards from node v_b looks at the rows of v_k's neighboring nodes and column D. The pheromone concentration update for entry (v_b, v_D) in v_k's RT is :

$$\varphi(v_b, v_D) = \varphi(v_b, v_D) + \frac{\epsilon}{T'} \tag{5.3}$$

where T' is $T(v_S, v_D) - T(v_S, v_k)$. This emphasizes more pheromone concentration on the path that is closest to the destination. All other neighboring node's pheromone concentration in column D are decremented as above (equation 5.2).

When the BANT reaches the source node S, the route is established from the source node S to the destination node D.

5.1.7 Benefit of source update

In the described methodology, the FANT performs the *source update* helping another FANT which considers the source as the destination to find the path easily. That is, if node v_i is the source for a FANT travelling to destination v_d, then the FANT updates the pheromone content of the source column in the RT of node (v_k) that it has just visited and selects the next hop by considering the entries in the destination column for the neighboring nodes of v_k. By doing so, the FANT selects the next node based on the best path that one of its neighboring nodes can provide. Also, updating the source column indicates the best available path that is reachable from the source v_i to v_k. When another FANT considers v_i as its destination and reaches node v_k, it always considers the destination column (v_i) in the RT of v_k to find the best path. The pheromone concentration in the entries on v_i column of RT indicates the best path to v_i through its current node's neighbors. Therefore, the *source update* technique improves the overall performance of the algorithm for finding a route.

5.2 Algorithm

Here is the algorithm for forward and backward ant.

```
Glossary:

 Stack: Keeps the visited nodes of the probable route.

 Totaltime: Keeps the total time elapsed after the forward ant left the
 source node.

 Visitedhop: Keeps the list of hops that are visited by the particular
 forward ant.

 Current: The node where the forward ant currently resides.

 Nexthop: The node that will be selected as the next hop for the
 forward ant.

 Prehop: The last node that was visited by the forward ant.


Procedure ANT(S,D) begin
    Stack ⇐ empty stack
    TotalTime ← 0
    Stack ⇐ (S, TotalTime)
    VisitedHop [S] ← 1
    Current ← S
    while (Current ≠ D)  /*Launch Forward Ant */
    begin
        NextHop ← ∅
        /* Look at all the unvisited neighboring nodes of
        CurrentHop using the routing table. That is look at
        column D and determine the next node of Current
```

```
    that produces the best path */
    if (Unvisited Node ⊆ N_current)
        NextHop ← Select an unvisited node
    else /*If all nodes have been visited then look at
                the pheromone  concentration (PH) in column
                D for all neighbors. Select the node with
                largest pheromone concentration. The greater
                pheromone value is considered as the next hop
                node only if the next hop node (NextHop)
                is not visited by the ant already. This
                information can be obtained from the ant's
                VisitedHop table. (Detects cycles).*/
    begin
        Find j where max_{v_j ⊆ N_current}(PH(v_j, D))
        and v_j is not visited by this ant
        if (v_j exists) NextHop ← v_j
    end

    if (NextHop ≠ ∅)
    begin
        PreHop ← Current
        Current ← NextHop
        TotalTime ← TotalTime + W(PreHop,Current)
        [Source Update: Update the routing table for
        NextHop using the PreHop and source information.
        Increase the pheromone value in entry (PreHop, S)
```

```
            by equation 5.1 and decrease pheromone value on all
            other entries in column S by equation 5.2 .
            This is a separate function]
            Stack ⇐ (Current, TotalTime)
            VisitedHop [Current] ← 1
        end
        else /* There is no path from current node to the
        destination node */
        begin
            Pop Stack
            /*Pop stack again to retrieve the last
            information*/
            (Current, TotalTime) ⇐ Stack
            if (Current = S) exit loop
        end
    end While

   (Current, TotalTime) ⇐ Stack

    While (Current ≠ S) /*Launch Backward Ant */
    begin
            PreHop ← Current
            (Current, TotalTime) ⇐ Stack
            T' = T(v_S, v_D) - TotalTime
            [Update routing table of Current node for
            destination node S using PreHop and T' using
```

	All Nodes in the Network	
Neighbor Nodes	PH	Visit

Table 5.1: Dummy routing table for explanation.

```
            equation 5.3]
            Stack ⇐ (Current, TotalTime)
            VisitedHop [Current] ← 1

    end While
end
```

5.3 Demonstration of Algorithm using an Example

Here is an illustration of the aforementioned algorithm with an example. Figure 5.1 shows a network of an eight node graph. Each link is associated with a time parameter. The explanation of the routing table used in the algorithm is given through table 5.1. In table 5.1, PH denotes the pheromone concentration and $Visit$ denotes the total number of times the ants have visited the neighboring nodes through this node.

Let us assume a FANT is moving from the source S (node 3) to destination, D (node 6). Also assume that the FANT has selected node 4 as its next hop ($NextHop$ = 4) link. Therefore, the current node is 4. At this point, the FANT performs the source update on node 4. The RT for node 4 before this update is shown in the

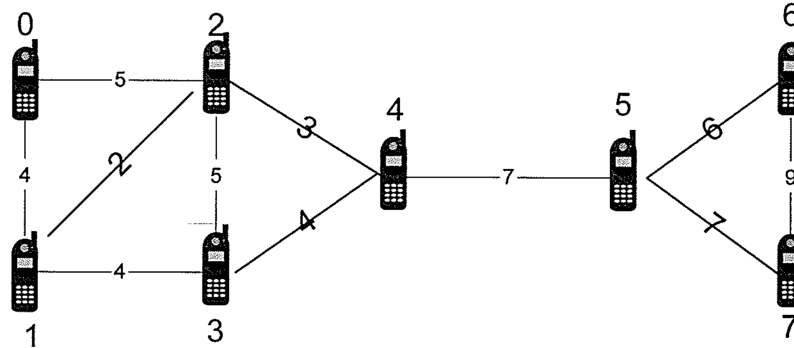

Figure 5.1: An Example of ad hoc network

Network Node	0		1		2		**3**		4		5		*6*		7	
2	1	1	97.6	8	111.7	5	**38.7**	**5**	0	0	0	1	*0*	*2*	0	1
3	42	7	1	1	11.7	4	**32.8**	**4**	0	0	0	0	*0*	*1*	0	1
5	0	0	0	1	0	0	**0**	**1**	0	0	140	10	*67*	*10*	64	10

Table 5.2: Routing table of node 4 before source update.

table 5.2. The FANT has memory of the source and the previous hop ($PreHop = 3$) node. It performs the source update by increasing the pheromone value on (3,3) from 32.8 to 57.8 and the *Visit* entry from 4 to 5. For all other entries ((2,3),(5,3)) it decrements the pheromone value. The RT after the update is also shown for node 4 in table 5.3.

At node 4, the FANT selects the next hop node to traverse. The FANT needs to reach destination 6. So, it looks at the destination column 6, of the RT at node 4 (table 5.3). From the of the node 4's RT it is found that all the neighboring nodes have been visited through this current node 4 to reach the destination node 6. Therefore, the FANT has to decide about the next hop using the pheromone concentration on

Network Node	0		1		2		**3**		4		5		*6*		7	
2	1	1	97.6	8	111.7	5	**27.1**	**5**	0	0	0	1	*0*	*2*	0	1
3	42	7	1	1	11.7	4	**57.8**	**5**	0	0	0	0	*0*	*1*	0	1
5	0	0	0	1	0	0	**0**	**1**	0	0	140	10	*67*	*11*	64	10

Table 5.3: Routing table of node 4 after source update.

Network Node	0		1		2		**3**		4		5		*6*		7	
4	24	6	45	7	55	7	**45**	**6**	84	6	0	0	*0*	*1*	0	1
6	0	2	0	2	0	1	**0**	**1**	0	1	0	0	*166.4*	*11*	6	3
7	0	1	0	1	0	1	**0**	**1**	0	0	0	0	*0.2*	*1*	82.9	10

Table 5.4: Routing table of node 5 before source update.

the links from node 4. The pheromone value in column 6, indicates the best path to choose. The pheromone value for the link from 4 to 5 is the greatest (67) among the links from node 4 to all neighboring nodes. Thus, it is assumed that node 5 may lead to a better solution. The FANT should take this node as a next hop neighbor.

When the FANT reaches node 5, it also performs source update as usual (table 5.5). The next task of the FANT is to choose the next neighbor to move to node 6. Now, it is obvious from the RT (table 5.4) of node 5 that the FANT will select node 6 as its next node towards destination node as the link between node 5 and 6 contains greatest pheromone value. When the ant moves to node 6, it realizes it has reached the destination. This will end the route discovery phase by the FANT.

The destination node 6 launches a BANT to the source node 3. The BANT will

Network Node	0		1		2		**3**		4		5		*6*		7	
4	24	6	45	7	55	7	**54**	**7**	84	6	0	0	*0*	*1*	0	1
6	0	2	0	2	0	1	**0**	**1**	0	1	0	0	*182.4*	*12*	6	3
7	0	1	0	1	0	1	**0**	**1**	0	0	0	0	*0.2*	*1*	82.9	10

Table 5.5: Routing table of node 5 after source and destination update update.

update the RT of each node on its path. When the BANT reaches node 5 from node 6, it performs update on the RT of node 5 that will facilitate the path to node 6. The pheromone concentration of the link is 166.4 as shown in the table 5.4. The BANT changes the pheromone concentration of that link to 182.4 according to the equation 5.3. The present status of the RT at node 5 is shown in table 5.5. The BANT does this similar update along its path and finally establishes the best path from source node to the destination node.

5.4 Summary

In this chapter, the description of the proposed algorithm based on the ACO technique for a dynamic application, routing in ad hoc network is given. The applicability of the algorithm is also shown by an illustrative example. In the next chapter, the implementation results of the proposed algorithm is presented.

Chapter 6

Experimental Result

In the last chapter, the proposed algorithm for routing in ad hoc network was described. The algorithm has been implemented on two different parallel computing platforms: distributed memory machine using MPI and shared memory machine using OpenMP. The performance of the algorithm on both these architectures is described in this chapter.

6.1 MPI Results

In this section, the performance results of the parallel ACO routing algorithm on a ten node network of workstations running MPI will be discussed. The input graphs were generated using a random graph generator. Given N, the number of nodes, and P, the number of processors, $\frac{N}{P}$ amount of data is distributed over the processors. For example, with 500 nodes and 10 processors, each processor is allocated 50 nodes per processor. Each ant, residing within a processor, determines the best route for each of the nodes in its subgraph to every other nodes. It computes the all-pairs best route for each of its nodes in the subgraph.

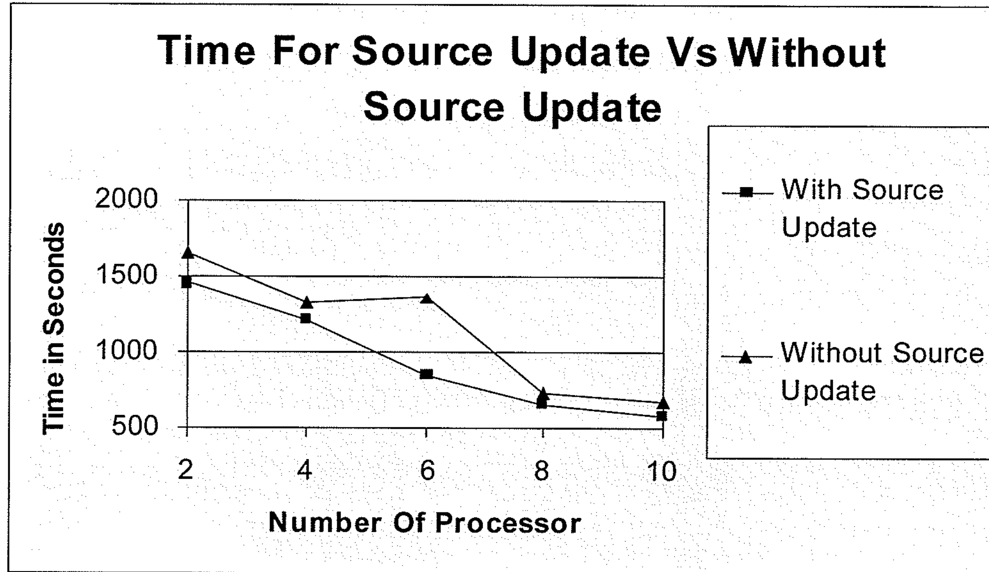

Figure 6.1: Performance results with and without source update

Figure 6.1 shows the scalability results with varying number of processors for both source update and without source update. From the figure it is clear that the algorithm converges faster to a solution using source update technique.

Figure 6.2 illustrates the scalability results with varying number of processors and nodes. As the figure indicates, the execution time decreases as the number of processors increases. It is assumed that the number of ants is equal to the number of processors, with one ant per processor.

The percentage of communication time versus the number of processors is shown in Figure 6.3. It is notable that there is faster convergence for larger nodes, thereby decreasing the percentage of communication with the increase of nodes in the network. Also, note that with the increase in data size, the amount of data per processor also increases, thereby increasing the computation overhead. However, the amount of computation per node is very fine-grained. That is, there is very minimal task per node. Therefore, the amount of computation is substantially lower than the

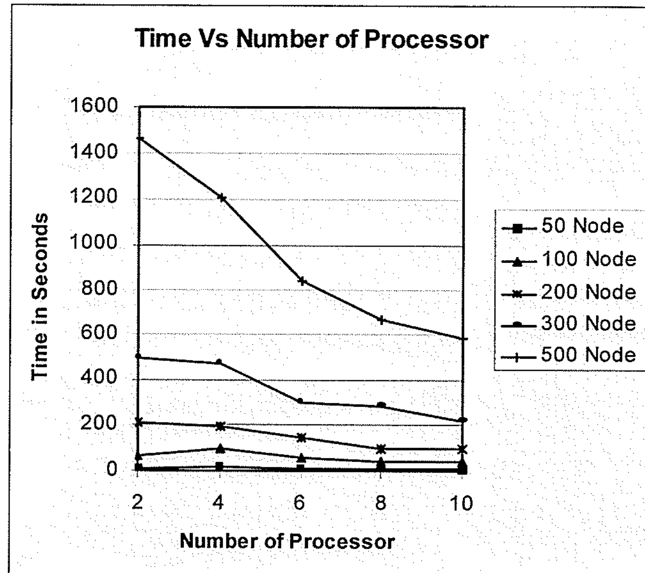

Figure 6.2: Scalability results with varying number of processors and nodes

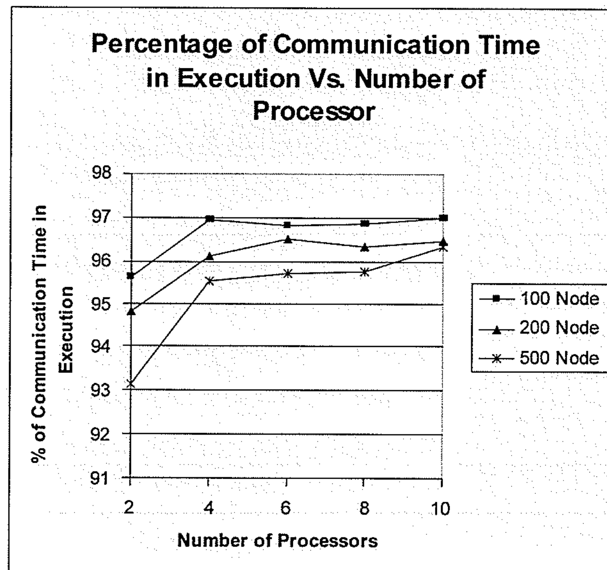

Figure 6.3: Number of processors versus percentage of communication time

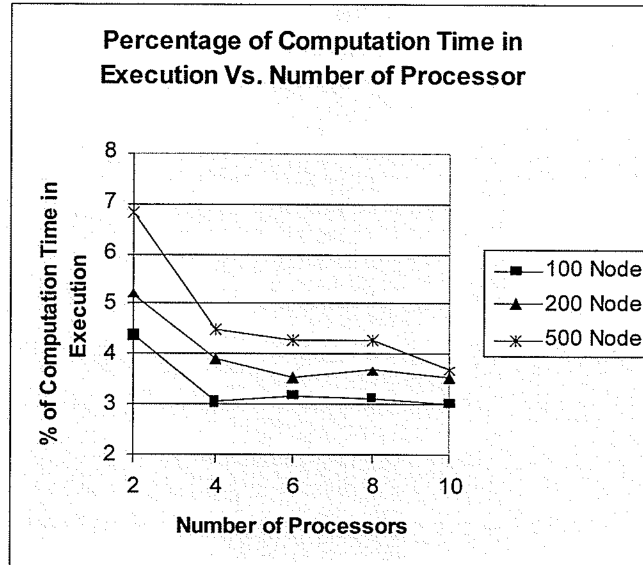

Figure 6.4: Number of processors versus percentage of computation time

communication overhead.

As can be seen from Figure 6.4 for 500 nodes, the percentage of computation is only 3.5% on 10 processors. For ad hoc networks, the communication time is the bottleneck. However, the results obtained from the implementation indicate that the algorithm converges faster with increasing number of ants with varying number of processors.

In Figure 6.5 the number of processors is fixed to ten while the number of ants is varied for 200 and 300 nodes network. The figure clearly illustrates a decreasing execution time demonstrating the scalability of the parallel algorithm. The best result is obtained when one ant is associated with each processor. This obviously distributes the workload among the ants obtaining a good load balance. With just one ant, the ant is overloaded with work in finding the routes for all pairs of nodes in the network.

The relative speedup of the algorithm is little over 7 for 10 ants (10 processors) as shown in Figure 6.6. Relative speedup is calculated as $\frac{Execution\ time\ of\ 1\ ant}{Execution\ time\ of\ 10\ ants}$. This

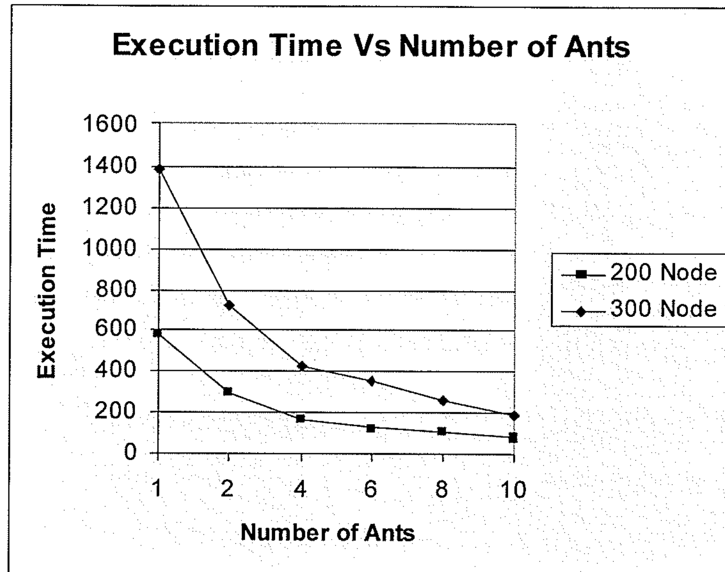

Figure 6.5: Number of ants versus Execution time with fixed number of processors and nodes

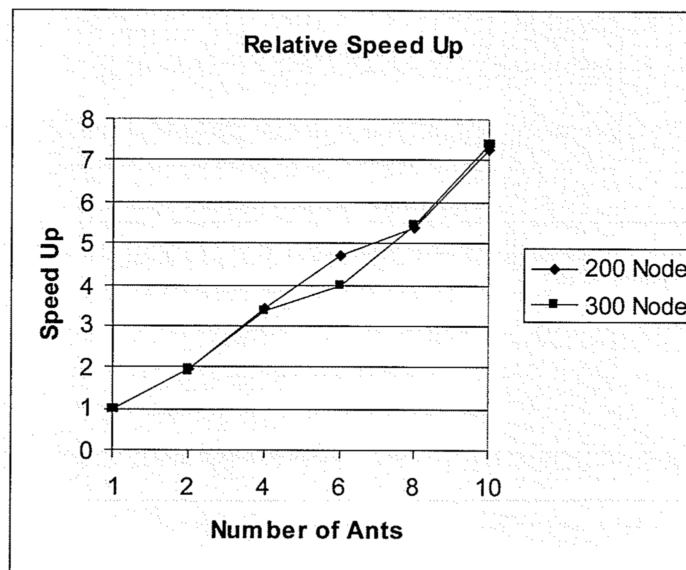

Figure 6.6: Speedup Results

speedup again illustrates the scalability of the algorithm.

6.2 OpenMP Results

In this section, the result that is obtained by running the algorithm on a shared memory machine is discussed. The shared memory machine contains eight multiprocessors and supports OpenMP for parallel computation. As the machine is not a distributed shared memory machine, the graph need not be partitioned and distributed to each processor.

The program uses *parallel* construct in OpenMP which creates a user specified number of parallel threads and divides the work among these threads (note that the ants represent the threads and are used interchangeably). There are three types of scheduling techniques that OpenMP provides: *static*, *dynamic* and *guided* scheduling [14]. We have experimented with each of these policies by varying the chunk sizes. Here, a chunk refers to the partition of data allocated to each ant. Each ant performs the necessary computations to find the route from a node to every other node. When one ant completes its task, it looks for another chunk of data.

In static scheduling, chunks are statically assigned to each thread. As the name implies, in the dynamic scheduling policy chunks are assigned to threads dynamically at run time in a round robin fashion. This distribution balances the load among the threads. In the guided policy, the first chunk size is maximum chunk size $(\frac{Number\ of\ Iteration}{Number\ of\ Threads})$ and each of the other chunk sizes decreases exponentially.

Figure 6.7 shows the scalability results for various number of ants in terms of execution time for various number of nodes in the network. Here, static scheduling technique is used. Therefore, the amount of workload is predefined. Each thread has equal amount of work load. The execution time decreases with the increase in

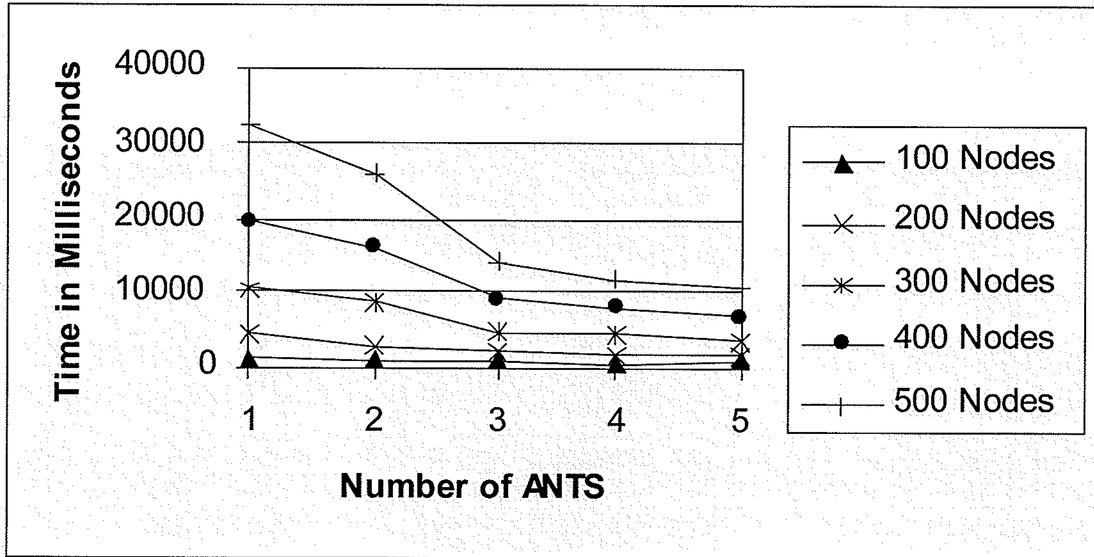

Figure 6.7: Scalability results for varying number of ants

number of ants. This implies that the ants cooperatively work together in solving the task of finding the best path from source to the destination in minimum time.

Figure 6.8 shows the scalability results for the three scheduling policies with 400 nodes. The maximum chunk size is allocated to each ant, which depends on the number of nodes and the number of ants ($chunksize = \frac{number\ of\ nodes}{number\ of\ ants}$). In static and dynamic policies, each ant is given an equal amount of work initially. Therefore, these two scheduling policies do not show much difference.

Figure 6.9 shows the graph with varying chunk sizes for 8 ants and 400 nodes with the different scheduling policies. The dynamic scheduling gives the best results. This can be explained as follows. The scheduler dynamically assigns chunks to each ant as they complete execution of the current chunk and balances the load among these ants. In static scheduling, the user allocates the chunk sizes to the ants. The best results is obtained when the chunk size is 25 or 50. In static scheduling, since the user intervenes, the load is not properly balanced. That is, though all ants have

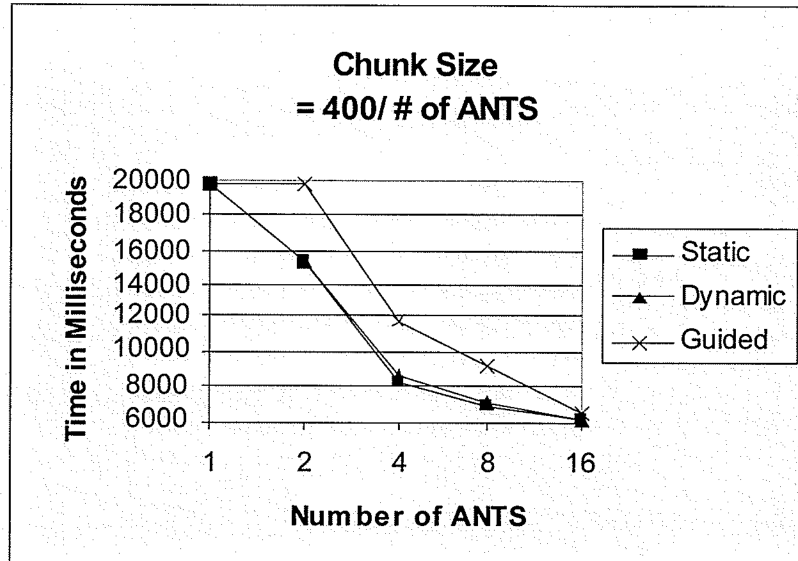

Figure 6.8: Scheduling policy results

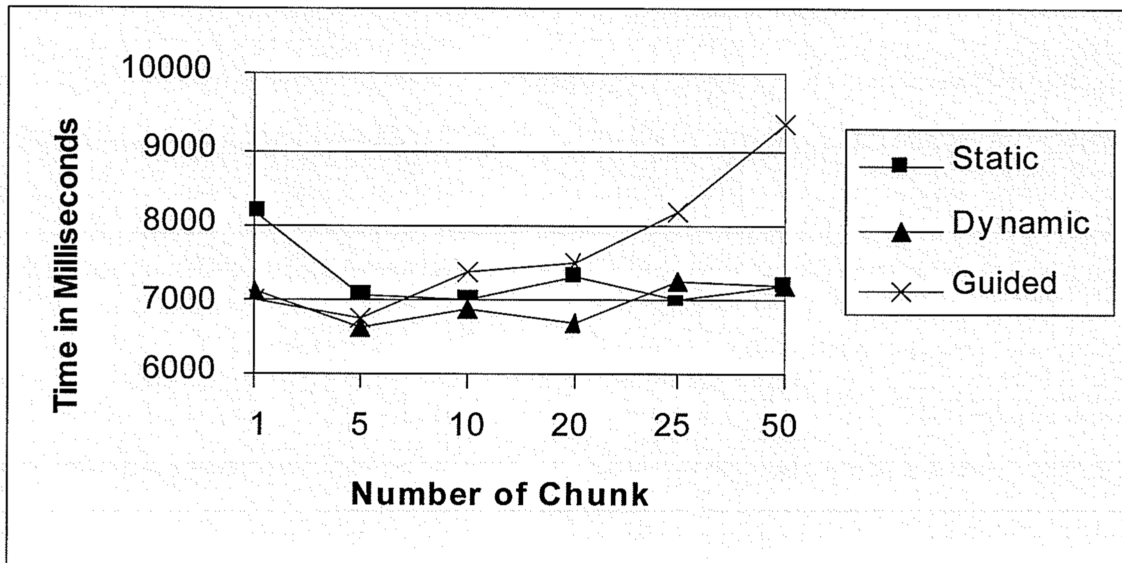

Figure 6.9: Performance Result with Varying chunk sizes for 8 ants and 400 nodes

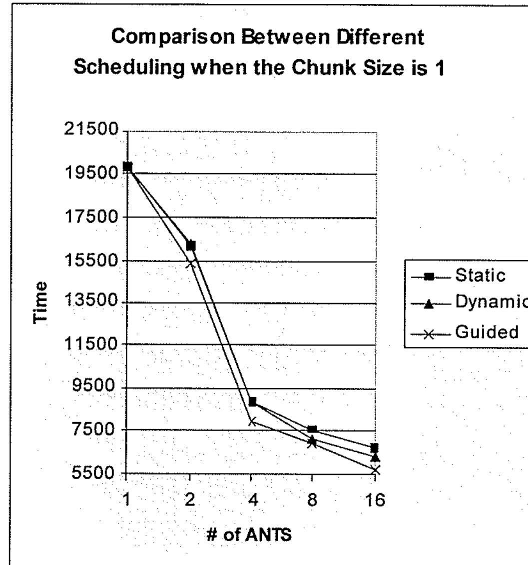

Figure 6.10: Performance results of the scheduling policies when chuck size is unspecified

the same amount of work load, some ants may have less or more computations and communications than others. This is also true for the guided scheduling policy.

In Figure 6.10 the chunk size is 1. Here the guided policy gives the best performance result. In the dynamic scheduler, each ant gets a chunk size of 1 whenever it runs out of work. However, in the guided schedule, the amount of load an ant receives depends entirely on the scheduler and it is not assigned in any order such as round robin as it is done in the dynamic scheduler.

6.3 Summary

In summary, the static scheduling policy gives the best result when the user properly load balances by defining the maximum chunk size. Dynamic scheduling policy is better for smaller chunk sizes than the maximum chunk size. The guided is best when the compiler schedules the load depending on the status of the ants, idle or

Number of Ants	Time in seconds	
	OpenMP	MPI
1	10.48	1384.47
2	6.18	719.03
4	4.52	426.03
6	3.91	344.25
8	3.37	254.59

Table 6.1: Comparison of OpenMP and MPI.

otherwise.

Table 6.1 compares the performance results of MPI [26] and OpenMP implementations. The algorithm runs extremely fast on a shared memory architecture than on a distributed memory architecture. In the distributed implementation, the number of ants is equal to the number of processors and an equal amount of data is allocated per processor. The obvious bottleneck in the distributed algorithm is the amount of communication. The amount of computation compared to communication is very minimal as shown in [26]. This communication problem in avoided in the shared memory architecture.

Chapter 7

Conclusion

This thesis presented a parallel routing algorithm for MANETs using the Ant Colony Optimization (ACO) metaheuristic search technique. The algorithm was implemented on a network of workstations running MPI and on a shared memory multiprocessor running OpenMP. The parallel algorithm based on source update scales well for varying node and machine sizes. We also developed a technique to detect cycles.

In the distributed memory machine, the number of processors and nodes was fixed while varying the number of ants in the algorithm. This produced a gradual decrease in execution time and the best result was obtained when the number of ants was equal to the number of processors. This resulted in a good load balance due to the even distribution of work among the ants. The relative speedup of the algorithm was little more than 7 for 10 ants (10 processors). We also noticed that the percentage of communication among the ants is much higher than the percentage of computation by an ant. MANETs are communication intensive applications. However, this does not degrade the performance of our algorithm indicating a fast convergence rate in finding the best paths.

In the shared memory environment, the algorithm was experimented with var-

ious chunk sizes for three different scheduling policies (static, dynamic, guided) in OpenMP. Each scheduler behaved differently depending on the chunk sizes. The static scheduler performed well when the user defines the maximum chunk size and allocated it to each of the ants. The dynamic scheduling policy was good when the chunk size was small. This was achieved by distributing work depending on the computation and communication ratio of each chunk. In the guided case the performance was best when the user does not predefine the chunk size and leaves it up to the compiler and scheduler to properly determine the load to be balanced among the ants.

When comparing the execution times of both implementations, it was found that the OpenMP implementation outperforms the MPI implementation for this problem. This is mainly due to the multithreading and shared memory environment, which removes the communication overhead present in the algorithm by overlapping computation with communication.

Chapter 8

Contributions

The contributions resulting from the this thesis are:

Journal	Mohammad Towhidul Islam, Parimala Thulasiraman and Ruppa K. Thulasiram, "Implementation Of Ant Colony Optimization Algorithm For Mobile Ad Hoc Network Applications: OpenMP Experiences", Journal of Parallel and Distributed Computing Practices, Special Issue on OpenMP: Experiences, Implementations and Applications,(Ed. Ami Marowka), Nove Science Publishers, 2003 (to appear).
Book chapter	Parimala Thulasiraman, Ruppa K. Thulasiram and Mohammad Towhidul Islam, "An Ant Colony Optimization Based Routing Algorithm in Mobile Ad hoc Networks and its Parallel Implementation", in High Performance Scientific and Engineering Computing-Hardware/Software Support, (Eds. Laurence Yang, Yi Pan), pp 267–284, Kluwer, (to appear).

Conference	Mohammad Towhidul Islam, Parimala Thulasiraman and Ruppa K. Thulasiram, "A Parallel Ant Colony Optimization Algorithm for All-Pair Routing in MANETs", IEEE Computer Society Proceedings of the Fourth International IPDPS Workshop on Parallel and Distributed Scientific and Engineering Computing with Applications (PDSECA 2003), April 2003, Nice, France.

Chapter 9

Future Work

Routing in an ad hoc networks is a new research area with many challenging unsolved problems. In this section, we highlight only some of the important work that can come out of this thesis.

- Parallel Processing: The algorithm has been implemented on a very small homogeneous machine. It would be interesting to extend this work to heterogenous computing since the nodes in an ad hoc network are heterogenous devices. This would seem to be a more realistic approach.

- Simulation: At this moment it is difficult to compare the parallel routing algorithm with other existing algorithms since there exist no other work on routing in ad hoc networks that has been parallelized. However, it would be possible to simulate the sequential version of the proposed ACO algorithm and compare to other existing routing algorithms. This is already underway. The simulation platform that is chosen for this is NS-2 [7] since all other existing routing algorithms have implemented on this platform.

- Energy Aware Routing: Another important area of research in ad hoc networks

that is gaining popularity is in energy aware routing protocols. The proposed ACO algorithm can be extended to include power aware routing. To implement this, the ACO equations have to be redesigned. The algorithm then has to incorporate these equations. This is definitely for future work since the work is quite involved.

Bibliography

[1] P. Arabshahi, A. Gray, I. Kassabalidis, A. Das, S. Narayanan, M.A. El-Sharkawi, and R.J. Marks II. Adaptive routing in wireless communication networks using swarm intelligence. In *Proceedings of the AIAA International Communications Satellite Systems Conference*, pages 17–20, Toulouse, France, April 2001.

[2] R. Arlauskas. ipsc/2 system: a second generation hypercube. In *Proceedings of the third conference on Hypercube concurrent computers and applications*, pages 38–42. ACM Press, 1988.

[3] Arvind and Robert A. Lannucci. Two fundamental issues in multiprocessing. In *4th International DFVLR Seminar on Foundations of Engineering Sciences on Parallel Computing in Science and Engineering*, pages 61–88. Springer-Verlag New York, Inc., 1988.

[4] Kenneth E. Batcher. Design of a massively parallel processor. *IEEE Transactions on Computers*, 29(9):836–840, September 1980.

[5] R. Beckers, J.L. Deneubourg, and S. Goss. Trails and U-turns in the selection of the shortest path by the ant Lasius niger. *Journal of Theoretical Biology*, 159:397–415, 1992.

[6] E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm Intelligence: From natural to artificial systems.* Oxford University Press, Oxford, 1999.

[7] Lee Breslau, Deborah Estrin, Kevin Fall, Sally Floyd, John Heidemann, Ahmed Helmy, Polly Huang, Steven McCanne, Kannan Varadhan, Ya Xu, and Haobo Yu. Advances in network simulation. *Computer*, 33(5):59–67, 2000.

[8] S. M. Brunett, J. Thornley, and M. Ellenbecker. An initial evaluation of the Tera multithreaded architecture and programming system using the C3I parallel benchmark suite. In *Supercomputing '98*, pages 1–19, Orlando, Florida, November 1998.

[9] Bernd Bullnheimer, Richard F. Hartl, and Christine Strauß. An improved ant system algorithm for the vehicle routing problem. *Annals of Operations Research*, 89:319–328, 1999.

[10] Bernd Bullnheimer, Gabriele Kotsis, and Christine Strauß. Parallelization strategies for the ant systems. In Renato De Leone, Almerico Murli, Panos M. Pardalos, and Gerardo Toraldo, editors, *High Performance Algorithms and Software in Nonlinear Optimization*, volume 24 of *Applied Optimization*, pages 87–100. Kluwer Academic Publishers, Dordrecht, Netherland, November 1998.

[11] Daniel Câmara and Antonio Alfredo F. Loureiro. A novel routing algorithm for hoc networks. *Baltzer Journal of Telecommunications Systems*, 18(1–3):85–100, 2001.

[12] Gianni Di Caro and Marco Dorigo. Antnet: Distributed stigmergetic control for communications networks. *Journal of Artificial Intelligence Research*, 9:317–365, 1998.

[13] Soumen Chakrabarti and Katherine Yelick. Implementing an irregular application on a distributed memory multiprocessor. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 169–178. ACM Press, 1993.

[14] Rohit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2001.

[15] A. Colorni, M. Dorigo, and V. Maniezzo. Distributed optimization by ant-colonies. In *Proceedings of the European conference on artificial life, (Ed. F. Verla and P. Bourgine)*, pages 134–142, Cambridge, Mass., 1991. MIT Press.

[16] NCUBE Corporation. Ncube users handbook, 1987.

[17] P. Delisle, M. Krakecki, M. Gravel, and C. Gagné. Parallel implementation of an ant colony optimization metaheuristic with OpenMP. In *International Conference of Parallel Architectures and Complication Techniques, Proceedings of the Third European Workshop on OpenMP*, pages 8–12, Barcelona, Spain, September 2001.

[18] Marco Dorigo and Luca Maria Gambardella. Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, April 1997.

[19] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics-Part B*, 26(1):1–13, 1996.

[20] Michael J. Flynn and Kevin W. Rudd. Parallel architectures. *ACM Computing Surveys*, 28(1):67–70, March 1996.

[21] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing*. Addison Wesley, second edition, 2003.

[22] William Gropp and Ewing Lusk. Why are PVM and MPI so different? In Jack Dongarra Marian Bubak and Jerzy Wasniewski, editors, *Parallel Virtual Machine and Message Passing Interface*, volume 1332 of *Lecture Notes in Computer Science*, pages 3–10, Crakow, Poland, November 1997. Springer.

[23] M. Güneş, U. Sorges, and I. Bouazzi. ARA: The ant colony based routing algorithm for MANETs. In *Proceedings of the International Conference on Parallel Processing Workshops*, pages 79–85, Vancouver, B.C., August 2002.

[24] Herbert H. J. Hum, Kevin B. Theobald, and Guang R. Gao. Building multithreaded architectures with off-the-shelf microprocessors. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 288–294, 1994.

[25] Mohammad Towhidul Islam, Parimala Thulasiraman, and Ruppa K. Thulasiram. Implementation of ant colony optimization algorithm for mobile ad hoc network applications: OpenMP experiences. *Parallel and Distributed Computing Practices (to appear)*, 2003.

[26] Mohammad Towhidul Islam, Parimala Thulasiraman, and Ruppa K. Thulasiram. A parallel ant colony optimization algorithm for all-pair routing in MANETs. In *IEEE Computer Society Proceedings of the Fourth International IPDPS Workshop on Parallel and Distributed Scientific and Engineering Computing with Applications (PDSECA 2003)*, Nice, France, April 2003.

[27] Micheal J Jipping and Gary Lewandowski. Parallel processing over mobile ad hoc networks of handheld machines. In *MobiHoc*, pages 267–270, Long Beach, CA, USA, 2001.

[28] David B Johnson and David A Maltz. Dynamic source routing in ad hoc wireless networks. In Tomasz Imielinski and Hank Korth, editors, *Mobile Computing*, volume 353, chapter 5, pages 153–181. Kluwer Academic Publishers, Boston, 1996.

[29] Alan H. Karp. Programming for parallelism. *IEEE Transactions on Computer*, 20(5):43–57, 1987.

[30] I. Kassabalidis, M.A. El-Sharkawi, R.J. Marks II, P. Arabshahi, and A. Gray. Adaptive-SDR: adaptive swarm-based distributed routing. In *Proceedings of the IEEE World Congress on Computational Intelligence*, Honolulu, Hawaii, May 2002.

[31] Tony Larsson and Nicklas Hedman. Routing protocols in wireless ad hoc networks - a simulation study. Master's thesis, Lulea University of Technology, Stockholm, 1998.

[32] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St Pierre, David S. Wells, Monica C. Wong-Chan, Shaw-Wen Yang, and Robert Zak. The network architecture of the Connection Machine CM-5. *Journal of Parallel and Distributed Computing*, 33(2):145–158, 1996.

[33] Leonid Oliker and Rupak Biswas. Parallelization of a dynamic unstructured application using three leading paradigms. *IEEE Transactions on Parallel and Distributed Computing*, 11(9):931–940, September 2000.

[34] Vittorio Maniezzo and Alberto Colorni. The ant system applied to the quadratic assignment problem. *Knowledge and Data Engineering*, 11(5):769–778, 1999.

[35] R. Michel and M. Middendorf. An island based ant system with lookahead for the shortest common subsequence problem. In *Proceedings of the Fifth International Conference on Parallel Problem Sloving From Nature*, volume 1498, pages 692–708, Berlin, 1998.

[36] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming: A POSIX Standard for Better Multiprocessing.* O'Reilly, 1996.

[37] Scott Oaks and Henry Wong. *Java Threads.* O'Reilly, 1999.

[38] Peter S. Pacheco. *Parallel Programming with MPI.* Morgan Kaufmann, San Francisco, 1997.

[39] Charles E. Perkins and Pravin Bhagwat. Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. In *Proceedings of the conference on communications, architectures, protocols and applications*, pages 234–244, London, United Kingdom, August 1994. ACM Press.

[40] Marcus Randall and Andrew Lewis. A parallel implementation of ant colony optimization. *Parallel and Distributed Computing*, 62(9):1421–1432, September 2002.

[41] E.M. Royer and C.E. Perkins. Multicast operations of the ad hoc on-demand distance vector routing protocol. In *Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM)*, pages 207–218, Seattle, Washington, August 1999.

[42] Ruud Schoonderwoerd, Owen E. Holland, Janet L. Bruten, and Leon J.M. Rothkrantz. Ant-based load balancing in telecommunications networks. *Adaptive Behavior*, 5(2):169–207, 1996.

[43] Charles L. Seitz. The cosmic cube. *Communications of the ACM*, 28(1):22–33, 1985.

[44] Thomas Stützle. Parallelization strategies for ant colony optimization. In A.E. Eiben, T. Bäck, M. Schoenauer, and H.-P. Schwefel, editors, *Parallel Problem Solving from Nature*, volume 1498 of *Lecture Notes in Computer Science*, pages 722–731, Amsterdam, 1998.

[45] Supercomputing Technologies Group, MIT Laboratory for Computer Science. *Cilk 5.3.2 Reference Manual*, November 2001. url: http://supertech.lcs.mit.edu/cilk/manual-5.3.2.pdf.

[46] Peter Tarasewich and Patrick R. McMullen. Swarm intelligence: Power in numbers. *Communications of the ACM*, 45(8):62 – 67, 2002.

[47] Parimala Thulasiraman, Ruppa K. Thulasiram, and Mohammad Towhidul Islam. *High Performance Scientific and Engineering ComputingHardware/Software Support*, chapter An Ant Colony Optimization Based Routing Algorithm in Mobile Ad hoc Networks and its Parallel Implementation, pages 267–287. Kluwer, 2004.

[48] C. K. Toh. *Ad Hoc Mobile Wireless Networks: Protocols and Systems.* Prentice Hall, December 2001.

[49] T. White. Swarm intelligence and problem solving in telecommunications. *Canadian Artificial Intelligence Magazine*, 1997.

[50] A.Y. Zomaya. *Parallel Computing: Paradigms and Applications.* International Thompson Publishing, Boston, MA, 1996.