To the Graduate Council:

I am submitting herewith a thesis written by Purnachandra Rao Surapaneni entitled "Parallel Best-First Search: Optimizing the Number of Expansions Per Iteration on Slaves." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

David C. Mutchler, Major Professor

We have read this thesis
and recommend its acceptance:

Accepted for the Council:

Vice Provost
and Dean of the Graduate School

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at The University of Tennessee, Knoxville, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that acknowledgement of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by Dr. David C. Mutchler, or in his absence, by the Head of Interlibrary Services when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature Swaparani Pillai

Date MAY, 89

Parallel Best-First Search: Optimizing the Number of Expansions Per Iteration on Slaves

A thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Purnachandra Rao Surapaneni

May 1989

Dedicated to my Wife

Sailaja

ACKNOWLEDGEMENTS

I wish to express my sincere gratitude to Dr. David C. Mutchler, my major professor and advisor, for his invaluable advice, inspiration, guidance and help during the period of this work. I am especially grateful to Dr. Mutchler for his untiring efforts in reviewing the manuscript of this thesis. I wish to sincerely thank Dr. Robin Cocket and Dr. Tom Dunigan for their advice and constructive criticism of this work and for serving on my thesis committee.

I would like to thank the Computer Science Department for preparing me for a professional career and for supporting me as a Graduate Teaching Assistant.

I express my sincere appreciation to my family for their continuous encouragement, love and support.

Finally, my deepest admiration and love to my wife, Sailaja, for her support, patience and understanding. She makes all of this worthwhile.

ABSTRACT

Several classes of problems can be solved by searching a graph. The best-first (BF) search algorithm can be used to generate a minimum cost solution. BF search iteratively selects the best node using heuristic information and then expands the best node. Parallel best-first (BF) search iteratively selects the K best nodes (beam search) where K is the number of processors available. These K nodes are expanded simultaneously (in parallel). The effect of the choice for how many expansions each processor should do per each iteration is investigated through experiments. The effect of different parameters on the optimal choice of number of expansions per iteration (E) is also illustrated. The parameters are number of expansions per iteration, number of slaves, heuristics, computation time (c_1) and communication time(c_2). A simple mathematical model is developed to estimate the computation time per each expansion (c_1) and communication time (c_2). Intuitively, E is directly proportional to communication time. This is discussed and results are presented. Using the traveling salesman problem as a testbed, this thesis investigates the optimal value for E as a function of communication and computation time.

Contents

List of Figures

Chapter 1

INTRODUCTION

There are many scientific, engineering, and other classes of optimization problems which need an exact solution. These include the traveling salesman problem, job shop scheduling, routing and theorem proving. Some of these problems can be solved by finding a solution path through a graph. For many applications, it is not practical to do an exhaustive search of the search graph (state space) to find the solution. Memory and time constraints limit the feasibility of doing an exhaustive search. To overcome these limitations, there must be an alternative search strategy.

The most popular and practical strategy to reduce the search time for solving a problem is using heuristic knowledge. This approach involves using knowledge about the problem domain to reduce the search time. The heuristic information is incorporated into the problem which is represented by a graph. There are many algorithms to implement this kind of search strategy (see, for example, [1], chapter 2). Some of these algorithms will not guarantee the optimal solution. They guarantee only a feasible solution. However, a few of the algorithms guarantee the optimal solution (see, for example, [1], chapter 2).

For any problem to be solved, it has to be represented precisely and systematically. It involves defining a problem state, the operators for exploration of the state, start and goal states. State definition contains the objective and possible configurations thus defining a problem precisely. One of these possible configurations can be used as a start state from which the process of problem-solving starts. There are operators that describe the actions governed by a set of rules. These operators lead to developing one or more paths from start to goal state(s). The goal states are the states which are acceptable as solutions to the problem. One or more of the developed paths from start to goal state(s) is taken as solution path(s) (Implicit graph). For example, in a traveling salesman problem, one of the cities can be selected arbitrarily as a starting city. The operators applied to this starting node expands this node which involves generating successors. If there are six cities numbered 1 to 6 and 1 is the arbitrarily selected starting state, then the successors for this starting node are 2, 3, 4, 5, and 6 (since these are unvisited cities). Repeating this process leads to generating a goal node.

A popular instance of the algorithms that guarantee the optimal solution is informed best-first search, some times called A* or BF [1, 2, 3]. As will be seen later, best-first search uses heuristic information systematically to guide the search as it progresses. The heuristic function applied to a node *N* projects the cost from node *N* to a goal node. For example, the nearest neighbor heuristic applied to a traveling salesman problem is the minimum of the costs from both the ends of the path already built to their nearest unvisited neighbor.

In the best-first (BF) algorithm, the estimated cost (f) is the sum of the actual cost along the path to the Nth node (g) and the heuristic value from the Nth node to the goal node (h). Its operation is as follows. It selects the best node depending on the projected cost (f). The best node is the one which is projected as the means of providing the least cost path. It then expands the best node. Expansion involves generating successors (children) by applying the operators to this node. It then computes g, h, and f values for the children. The process is repeated until the best node is a goal node. Each time a best node is selected (repeat loop), it is designated as one iteration of BF.

Thus the heuristic function is critical in two ways. First, BF finds the optimal solution provided the heuristic function is admissible. The heuristic function is said to be admissible if it never overestimates the actual cost for minimization problems. Second, it is critical in determining the computational efficiency. The better the function, the less the computational time [2]. This depends on how well the knowledge about the problem domain can be incorporated into the heuristic function. For a detailed study of this algorithm, refer [2,4].

The best-first algorithm described above is for a single processor. Its performance has been investigated by several people [4,5]. Recently, however, there has been a growing interest in implementing this algorithm in parallel. Understandably, this attention comes from the need to solve problems at better speeds. In beam search, we repeatedly select the best node (using the heuristic function) and expand it. One way to implement the beam search process in parallel is to select the K best nodes, instead of just the single best node, where K is the

number of processors available. These K nodes are then expanded simultaneously (in parallel). Usually this reduces the solution time. Even though this parallelization can be achieved on many kinds of machines, recently attention has been focussed on parallel computation on a network of small computers [6]. The reason for this kind of distributed concurrent computing is the low cost and availability of high-speed workstations and high speed local area networks.

There are a variety of implementation techniques for parallel best-first search depending upon the type of machines used [6,7]. In this thesis, a set of SUN workstations running on a local area network (LAN) are used for distributed processing. The SUN workstations run the BERKELY UNIX operating system and are connected by an Ethernet. Choosing a set of small computers over a main frame with multiple processors is appropriate because of the increasing importance given to this type of distributed concurrent processing in the real world, especially in research facilities. Many work stations in the computer labs are idle during the non peak hours. If they are network connected, these computers can be effectively used during this non peak time as loosely coupled multiprocessors to solve big problems.

There are two main models of distributed concurrent computing. One is the synchronous master-slave model in which a master controls the slaves (slaves or processes). The second is the asynchronous broadcast model in which all processes are considered equal. In this thesis, the synchronous master-slave model is used for reasons discussed in chapter 3. In the synchronous master-slave model, the master maintains the open list of nodes. From the open list, it selects the best

nodes and sends them to slaves for expansion. It then receives a set of newly generated nodes from each slave. This set was generated by expanding the node that was sent to the slave. The master sends the next set of nodes to slaves only after it had received the response for its previous set from all slaves. In other words, before it can send the next set it must have received all the newly generated information from all the slaves for its previous set and this information must be updated in the master. A distributed processing utilities package (DPUP) is used to establish the communication between the master and slave process [8]. The important parameters in the master-slave model are the number of processors (slaves), communication time between the master and slave, the heuristic function, and the number of expansions slaves make before sending back the newly acquired information to the master. Throughout this thesis, we call this last parameter E, the number of expansions per iteration. Every time the master sends nodes to its slaves, it is labelled as an iteration. The performance of parallel best-first search depends on these parameters, as well as the problem it is given.

It is important to note the effect of these parameters on the execution speed. The number of iterations usually decreases or stays constant as the number of expansions per iteration (E) increases for a given number of slaves. Decreasing the number of iterations does not necessarily mean increased speedup. It also depends on the time for each expansion and communication time. But both communication time and expansion time are fixed for a given problem. The most relevant parameter that can be varied under software control is the number of expansions per iteration. This leads to the following discussion.

The complete control of the parameter E (number of expansions per iteration) by software raises an important question. What is the optimal number of expansions per iteration for a given number of processors, given heuristic and fixed communication time? This is an appropriate question since E is the only parameter that can be varied solely in software. Intuitively, the answer depends on the ratio of communication time to computation time for expansion. If the communication time is small, the number of expansions per iteration should be less. This results in more informed search because the information is updated frequently. This may reduce the number of nodes expanded which in turn reduces the search time. It leads to higher speedup. On the other hand, if the communication time is high, the number of expansions per iteration should be greater. Otherwise, the slaves have to sit idle for a long time. The search time to solve the given problem may increase. This in turn reduces the speedup. It may be argued that the slaves may not be idle, and they may be working on another process. This argument can be countered by pointing out two flaws. First, most of these machines are dedicated machines and run just one process. And also, as pointed out before, these may be used during non peak hours for parallel processing. Second, speedup is the main objective of parallel processing, not the machine efficiency especially when used in a network of small computers.

This thesis answers the question raised in the previous paragraph through experiments. The effect of different parameters such as number of processors is analyzed. Empirical relations are developed for communication time and expansion time. Times are measured using these two parameters and are compared at

different number of expansions per iteration. There are graphical presentations between different parameters.

The traveling salesman problem is used as a testbed. A large number of problem instances are generated over a number of parameters with a wide range of values. The parameters include number of processors and number of expansions per iteration. We measure the average time taken to solve a problem for a given setting of the parameters. The effect of the parameters, especially E, on speedup is measured. The results are recorded and presented via graphs.

In sum, this thesis makes four contributions. First, it shows through experiments that the choice for how many expansions to do per iteration can make a significant difference in the performance of parallel best-first search. Second, it suggests how certain parameters affect the optimal choice for how many expansions to do per iteration. Third, it provides a simple mathematical model to estimate time per each expansion (c_1) and communication time (c_2). This mathematical model can be used as a rough predictor to obtain the optimal value for number of expansions per iteration (E). Fourth, it provides a large set of software for use in further experiments with parallel best-first search on a network of Suns.

Chapter 2

LITERATURE SURVEY

A fairly extensive literature survey was done during the course of this thesis. A computer search was done to obtain the pertinent literature. The following discussion briefly summarizes the analysis of this literature survey.

The attention on the development of efficient search procedures was started in the early 60's. Hart etc. al. [2] developed a formal basis for the heuristic determination of minimum cost paths. Their paper describes how heuristic information from the problem domain can be incorporated into a formal mathematical theory of graph searching and demonstrated the optimality property of a class of search strategies. One of the algorithms discussed in their paper was best-first (BF) search or A* for a single processor. They also discussed the evaluation function $f = g + h$ where g is the actual cost of a path from the start to node N and h is the projected cost from N to a goal node.

In their recent paper [4], Dechter and Pearl discussed the BF strategies and the optimality of A*. They also discussed the performance of A* for a single processor.

It can be seen from the literature that there has been a growing interest in parallel computation in this decade. Several people investigate different parallel algorithms. Trienekens [6] discussed parallel branch and bound on MIMD systems. In his paper, he classifies parallel branch and bound algorithms and develops a class of asynchronous branch and bound algorithms for execution on an Multiple Instruction Multiple Data (MIMD) system.

A lot of work is done in analyzing the performance of parallel computation. In their paper [9], Kindervater and Trienekens discussed the computational results concerning the solution of Knapsack, shortest paths and change-making problems by branch-and-bound, dynamic programming, and divide and conquer algorithms on three different computers. In their paper [10], Li and Wah developed sufficient conditions to guarantee that parallelism will not degrade the performance. They also showed that parallelism can have a speedup greater than the number of processors. They also present some anomalies owing to parallelism. In their discussion, Li and Wah also mention that deceleration anomalies are usually unavoidable and diminishing returns occur when the number of processors is large. They conclude by saying that best-first search is "a good search strategy that can guarantee little degradation in performance with increasing degree parallelism." This last conclusion is based on the assumption that there is no communication time involved.

There has been a discussion on the upper bound for the speedup of some parallel algorithms. In their paper [11], Quinn and Deo derive an upper bound for the speedup obtained by any parallel branch-and-bound algorithm using the best-

first search strategy. However, they confirm that parallel branch-and-bound can achieve nearly linear or even super-linear speedup under appropriate conditions.

The performance of three hypercubes was discussed by Dunigan in his technical report [12]. Dunigan has also developed a message-passing multi-processor simulator [13]. This simulator provides a multitasking environment for the development of algorithms for parallel processors using either shared or local memory.

Most of the earlier parallel work is done on main frame computers with multiple processors. However, there has been a growing interest on the use of network of small computers for parallel computing. Trienekens [6] discusses the need of such a parallel computation. In his paper [14], Vornberger discusses a method to solve combinatorial optimization problem on a multi-processor network. The paper also discusses the actual implementation of a fault tolerant algorithm on a network of computers. In their paper [15], Finkel and Manber discuss the implementation of back tracking of a distributed system. They developed a library package for tree-traversal algorithms such as backtrack and branch and bound that is implemented on a multi-computer (see [16], chapter 6, for explanation of these algorithms). They extensively discuss the tools for writing distributed programs for specific applications, dynamic division and distribution of work, minimum requirements from the operating system and fault tolerance.

Dunigan has developed a hypercube simulator on a local area network [17]. This package uses a library of message passing routines and multiple user processors to provide an environment for the development and testing of algorithms for hypercube parallel processors simulated by a collection of computers on a local area network. Each node of the simulated hypercube is a workstation on the network.

Most of the work done on the parallel computation discusses the speedup relative to the number of processors. In parallel computing, traditional speedup is defined as the time taken when the number of processors is one divided by the time taken when the number of processors is P. This thesis examines the effect that the number of expansions per iteration by the slaves has on speedups. The definition of speedup in this thesis replaces the word "processors" in the traditional speedup definition by the phrase "expansions per iteration by the slaves."

The traveling salesman problem is also extensively studied in the literature. Almost all the parallel algorithms are tested using a traveling salesman problem. Trienekens reports results for the traveling salesman problem [6]. The traveling salesman problem and minimum spanning trees were discussed by Held and Karp [18, 19]. They discussed theoretical aspects and different approaches to explore the traveling salesman problem. They also discuss the relation between the symmetric traveling salesman problem and the minimum spanning tree heuristic. Mohan [7] solved the traveling salesman problem using two implementations based on the branch and bound algorithm. One of the

implementations is synchronous and the other is asynchronous. He also discusses the speedups.

Chapter 3

EXPERIMENTAL DESIGN

3.1 The Code for Distributed Best-First Search

Informed best-first search, also known as A*, is perhaps the most popular search method that is both efficient and accurate. The definition of an efficient algorithm varies depending on the problem to be solved. In the case of A*, it may be defined as efficient since it considerable lesser time than the algorithms that do exhaustive search. A algorithm may be defined as accurate if it always gives the best solution possible. This method uses heuristic information to reduce search time. The algorithm can be adapted to distributed processing with few alterations. The strategy of ordinary best-first search is to use heuristic information to select the best available node for further search. The same strategy can be applied to distributed processing. Instead of selecting the single best node for expansion, the K best nodes are selected (beam search), where K is the number of processors available. This strategy can be applied to large classes of problems such as scheduling, routing, theorem proving, etc [2,3].

The accuracy and time that it takes to solve a problem using the best-first algorithm depends on the heuristic and the pruning mechanism used. The heuristic

function helps in selecting the node that appears to lead towards the goal. The operating mechanism of best-first search is described below.

The algorithm searches an implicit directed graph that consists of a set of nodes which represent the problem. In a directed graph, the nodes are joined by edges or arcs. Each node represents the problem state which consists of the actual cost along the path to this node, projected cost from this node to the goal node, a link to its parent and a set of links to the nodes that were generated from this node (child links). The parent link permits tracking the path back from the goal node to the start node once the goal is found. The child links are useful to keep track of the duplicates and to adjust the parent link if a better path is found. In addition to these, two lists, open and closed, are needed to keep track of the nodes yet to to be examined and already examined, respectively. The nodes that are already generated but not expanded are kept in the open list. The term expanded means generating the successors. The nodes that have been already expanded are kept in the closed list. A heuristic function is used to estimate the cost from the current state to the goal state(h). The generated nodes contain this heuristic estimated cost (h) and the actual cost from the initial state to the current node(g). The algorithm selects the best node depending on the combined cost (f) which is the sum of g and h. The best node is the one which has the lowest combined cost (f) among the nodes in the open list. In case of parallel best-first search, where there may be a need to select more than one best node, the best nodes are selected from the lowest f upwards.

The master selects the best nodes from the open list and sends them to the slaves. Each slave does E expansions beneath the node it is given and sends back the

subtree of generated nodes to the master. The information sent by the slaves includes f, g, and h values. The size of the data is approximately equal to 4(N+E(2(N-1)+4)) bytes. The master receives the newly generated information from the slaves. This information includes newly generated children, f, g and h values for the children and parent links. The master then inserts these new nodes in the lists appropriately. Insertion requires checking for duplicates. For each successor received from the slaves, the master checks both the open and closed lists to see whether a duplicate of this exists. If it already exists in any of the lists, then it is checked to see whether the new path is cheaper than the existing path. If the existing one is cheaper or the same, then throw the new one away. If the new one is cheaper, update the old one's parent link, f, g, and h values with the new node values. If a duplicate is found in the closed list and replaced with the new values, then move the node to the open list. If there is no duplicate, then insert the new node in the open list. Repeat the procedure until the goal is found or the open list is empty.

The goal is reached when the best node picked is the end of a complete path. If no goal is found and the open list is empty, then the algorithm terminates as a failure. One important point to note here is that if the slaves do more than one expansion per iteration, they have to do the whole procedure as the master does. The procedure includes checking for duplicates, goal nodes and pruning if necessary.

3.1.1 Communication via DPUP

In any distributed processing, efficient communication between the processors is extremely important. The efficient communication may be defined as sending information as quickly as possible without loosing any information. That goal is achieved in this research by using a utilities package written in the C language. This package, known as Distributed Processing Utilities Package (DPUP), was developed by the Department of Computer Science, University of Colorado, Boulder [8]. The DPUP package supports distributed concurrent computing on a local area network (LAN) of computers. Even though the master-slave model is selected in this thesis, DPUP also supports the broadcast model. In the broadcast model, there is no master control. All processes have a common communication channel. Using this channel, any message sent by a process can be heard or received by all other processes. The broadcast model is asynchronous. Both models have their own advantages and disadvantages. The main functions of this package are to establish communications between the processes, to send and receive data between them, and to create and terminate remote processes.

In this thesis, a master-slave model is used. Even though it is basically synchronous because the master controls the flow, it can be implemented as either a synchronous master-slave model or an asynchronous master-slave model. For both models, the master maintains the open list of nodes. From the open list, the master selects the best nodes and sends them to slaves for expansion. It then receives a set of newly generated nodes from each slave. This set was generated by expanding the node that was sent to the slave. In the synchronous model, the master

sends the next set of nodes to slaves only after it has received the response from all slaves for its previous set. In other words, before it can send the next set, it must have received all the newly generated information from all the slaves for its previous set of nodes sent, and this information must be updated in the master. In the asynchronous model, the master sends the next best node to a slave for expansion immediately after it receives information from that slave. In this project, the synchronous master-slave model is implemented.

In the synchronous master-slave model used in this project, the master controls the complete search operation for finding the minimum cost solution. The DPUP server processes (dp_server) are started first on both master and slave machines. The master then establishes the communication with all the slaves. All the communication is done through the master only. This point to point communication is based upon the stream sockets. There can be more than one process running on any slave. The master has the ability to start and create a remote process, send data to a process and receive data from a process, kill, close and exit a remote process and display the status operation.

The parallel best-first search involves the following. After the master establishes the communication, it creates a few starting states and puts them in the open list. As explained in the previous section, the master selects a set of best nodes. The number of nodes in the set is the smaller of the number of nodes in the open list and the number of slaves. Before sending the first set, the master may send some data describing the input problem to all the slaves. This information will be needed by the slaves for the execution of slave (remote process) routines.

This involves sending the cost matrix in case of traveling salesman problem. Once the nodes are sent to all slaves, the master keeps polling the slaves to check whether a slave is ready to send the newly expanded nodes. As explained before, the master inserts the new nodes in the lists. The master also checks for new solutions (i.e., any goal found). If any of the slaves returns a solution, it is checked against the existing solution if one exists. The better one is retained. If the solution value is improved, pruning is done on the open list with the new f. Pruning involves removing the nodes from the open list whose value is greater than or equal to the existing solution f value. Pruning is done only once for each iteration before selecting the new set of nodes from the open list. If the open list is empty and no goal is found, the algorithm returns as a failure. Before terminating, the master writes the results to the disk and terminates the remote processes by closing the stream sockets.

The synchronous slave model has many advantages compared to broadcast model. It reflects the natural style of the parallel algorithm. The mechanism is relatively easy to understand. Thus, this model appears to be more suitable for many distributed concurrent computations. Also, the model is easier to design than the broadcast model. The model, however, is not without any disadvantages. Since it is partially synchronized, some slaves may remain idle for some time. This waiting is due to the fact that the master does not send a new set until it receives data from all of its slaves working on the same problem set. This idle time may introduce some inefficiency in the algorithm especially in applications where such waiting has no advantage at all. This waiting may have some worth in problems like the traveling salesman in which more information speeds up the search. The

other disadvantage is the load on the master. Since the master controls all the slaves, this may lead to a bottleneck. The partial remedy for this disadvantage may be creating submasters which do some processing before sending data to the master. This remedy may also help in resolving another disadvantage which is the software limit on the number of processes that could be connected to a master. This limit is 7 in the case of DPUP.

3.1.2 Memory Management - Fibonacci Heap

Beam search selects the best node for each slave for each iteration, repeatedly. In order to do this efficiently, it is important to have good memory management in maintaining the open list. A new data structure for implementing the open list was proposed by [20]. These open list structures, termed Fibonacci heaps (F-heaps), are a modification of the binomial queues proposed by [21].

Each entry in a F-heap consists of a set of items, and one of these items is a pointer to a spot in the hash table. The hash table contains all the nodes. The data structure uses the f value ($g + h$) as its key. The key is used to pick up the best node. This involves picking up the node which has the lowest f value. If two nodes have the same lowest f value, then the one of those lowest f values nodes is picked arbitrarily.

The F-heap permits the following operations: *insert*, *find min*, *decrease key*, *delete*, *delete min*, and *meld*. *Insert* inserts the new item into the heap. If no

heap exists, it creates one. *Find min* finds the minimum node. For this purpose, it uses the value of *f* which is the sum of *g* and *h*. *Decrease key* decreases the value of a key by some non negative number. This situation arises when duplicates are found. If the just completed state has lower value than the existing one, then the existing *f* is decreased by the amount equal to the difference of two *f* values. If the new *f* is greater than the existing one, the new state is ignored. *Delete* deletes the existing state and associated values. This occurs only if at least one solution is found. Once a new solution is found, all the items that have the same or higher *f* value are deleted. *Delete min* deletes the minimum *f* value (the best node), which will be sent to a slave for further expansion. *Meld* merges two disjoint heaps into one and destroys the two original heaps.

In their paper [20], Fredman and Tarjan proved that F-heaps are asymptotically faster than binomial queues. They showed that the amortized times for the operations are: O(log n) for *delete min*, O(1) for *insert*, O(1) for *decrease key*, O(1) for *delete* (unless the deleted node is minimum).

3.1.3 Source Code

The code consists of three parts: Master routine, a slave routine, and a header file. The header file consists of slave names, number of nodes and general information such as problem size. This information is needed for both the master and slave routines. Both master and slave routines are large and complicated. Establishing the communication between the master and slaves, terminating the

slaves, and closing the sockets properly was the toughest part of the code as far as communications are concerned. Part of this can be attributed to the existence of errors in the documentation of DPUP. It was time consuming since it involves changing some system protected values. For this, system maintenance personnel help was needed. Finally, when it was ready, the experiments had to be conducted during the limited hours of 12 mid-night to 8:00 A.M. to avoid any interruption of student's lab work and other research activities.

Each of the master and slave routines are almost 1000 lines of C code. With the speed of F-heap and hashing array, it was possible to solve many problems quickly. As discussed in the results section, the time that it takes to solve a problem varies considerably depending on the heuristic. Even with efficient memory management, it was not possible to solve problems with large a number of cities. This is because as the number of cities increases, the number of states grows exponentially. This creates a dynamic memory problem. The accuracy of the program was thoroughly checked with a small number of test problems.

3.2 The Testbed Problems

The problem that has been used in this thesis is the traveling salesman problem. The traveling salesman problem involves finding a minimum cost tour that visits all N cities once and only once, and then returns to the start city. The cost matrix specifies the cost of travel between each pair of cities.

The traveling salesman problem is selected because it has been studied extensively. The reason for this could be because it is easy to understand but still a computationally complex problem. It is NP-complete [7, 18, 19]. It also has real world applications such as Vehicle routing, Job sequencing and Computer wiring [16]. Since it is computationally intensive and generates quite a bit of data for each expansion, it is a good problem for the study of computational and communication speedup, which is the main goal of this thesis.

The algorithm is tested by solving some randomly generated instances of the symmetric traveling salesman problem. One way of generating random problems is using the Euclidean model. In this, points are generated randomly and independently in a two dimensional space. Each point represents a city. The cost between two cities is the euclidean distance between the corresponding cities. Another way of generating the test problems is by using random graphs. In this method, the costs are drawn independently at random from a uniform distribution. The latter technique is used in this thesis to generate the problems randomly. The uniform distribution range is from 0 to 1000.

3.3 The Experiments

As mentioned in chapter 1, the important parameters that effect the performance are the number of slaves, number of expansions per iteration (E), the communication time, the computation time (time for each expansion) and the

heuristic. There is no software control on communication time. The computation time usually depends on heuristic but once the heuristic is fixed, there is no software control on this parameter. The number of slaves is bounded by the available hardware or by operating system limits. The only remaining parameter is E, the number of expansions per iteration. The optimal choice for E depends on the computation and communication times. With these factors in perspective, the experiments are designed.

In our experiments, the parameters that are varied are the number of cities, the number of slaves, the number of expansion slaves would make before sending back the results to the master (E), and the heuristic. The number of cities varied from 6 to 16; however, we report results only for 10-city problems. Memory limited the size of the problems we could consider. The memory requirement is about 2^N nodes in the worst case. Each node requires about 4(N+5) bytes. The number of slaves was set to 1, 2, 4 or 7. E is varied from 1 to 25.

The two heuristics used in the experiments are nearest neighbor and minimum spanning tree. They both use completely different strategies in computing the heuristic value. In the nearest neighbor heuristic, the next city is selected by looking at all cities not yet visited from both ends of the path already built. It then selects the closest city. In the minimum spanning tree heuristic, two trees are built edge by edge with all unvisited cities from both ends of the path. Edges are selected for inclusion in the tree in increasing order of their costs. It also makes sure that no cycle is formed. If any node appears more than once in a

path of a graph, it is called a cycle. The tree that gives the least cost is selected. Thus, in both the heuristics, the heuristic is computed from both the ends of the path already built (visited cities). The minimum of the two costs is taken for the next move. It should be noted that both the heuristics are admissible (underestimates costs).

For each setting of the parameters, we ran 100 random problem instances. For some settings, we ran 1000 problem instances. For each problem, we recorded the settings of the parameters, the number of iterations it takes to compute the exact solution, CPU time (both the system and the user), and real time elapsed. These results are later extracted for computational analysis.

Chapter 4

RESULTS

The computational experience with the parallel best-first algorithm is presented in this chapter. Graphs are presented wherever appropriate. The algorithm is applied to several sets of randomly generated problems. Even though the problems tested vary from 6 cities to 16 cities, results are presented only for 10 cities. Even though several issues are discussed, the focus is on the question of how to find the optimal number of expansions per iteration and how it is affected by the computation time for each expansion (c_1) and communication time between the master and the slaves (c_2).

Results are presented for two measures of time over 2000 problems. One measure is the average real time taken directly from the data. This is the actual elapsed time. The other measure of time is predicted elapsed time, based on the average number of iterations and also on the communication time (c_1) and computation time (c_2), averaged over a large number of problems with various number of slaves and expansions per iteration. The details are given in section 2 of this chapter. Results are presented and discussed for several parameters. The parameters are the number of expansions per iteration, the heuristic used, and the number of slaves. Most of the results are displayed in graphical form. The

maximum and minimum elapsed time that a problem takes in the set is also presented for a few sets of problems.

4.1 Real Time Versus Number of Expansions

The parameter of greatest interest in this study is the number of expansions per iteration (E). Figure 1 graphs this parameter versus real time (elapsed time) as obtained from the data. The number of slaves is either 1, 2, 4,or 7, and E varies from 1 to 25. There are two reasons for limiting the number of slaves to 7. One, DPUP allows only 7 slaves for each master. Second, performance of all SUN workstations is not same in the lab where these experiments are conducted. The reason for limiting the number of expansions per iteration to 25 is because it was observed for all cases considered for not more than 7 slaves, the optimal point is less than or equal to 16. One hundred problems are solved using the minimum spanning tree heuristic, and the average time curves are presented. As expected, the time drops initially as E increases and then starts rising. As can be seen from the graph, this trend is consistent with different number of slaves presented even though the value for E at which the time starts increasing differs. It can also be observed from the graphs that there are some irregularities in the curves. For example, there is a steep increase in time when the number of slaves are 1, 2, or 4 and at E equal to 4. The same problem set was rerun to check the accuracy when the number of expansions per iteration (E) is 4. Similar times are repeated. This irregular rise in time could be attributed to the fact that the

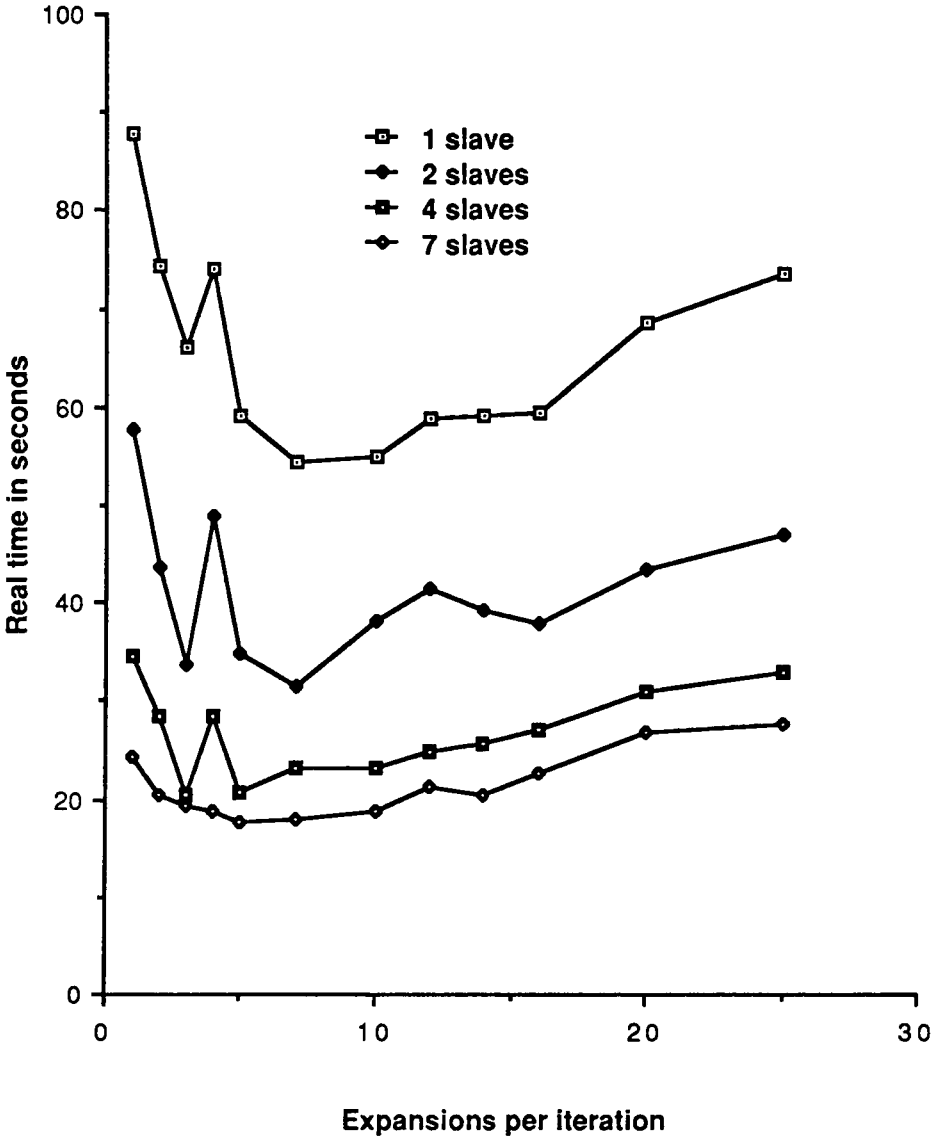

Figure 1. Average real time as a function of number of expansions per iteration.
Heuristic: Minimum spanning tree.

Ethernet uses packets to send the messages. There may be an uneven split of packets at that point which might raise the communication time.

We define the *speedup* at e to be time when the number of expansions per iteration is 1 divided by the time when the number of expansions per iteration is e. Note that this definition is not the same as the traditional definition for speedup in parallel processing. Figure 2 presents the speedup chart for the timings presented in figure 1. The optimal number of expansions per iteration is the number at which highest speedup is achieved. In figure 2, the speedup rises steadily in the beginning and then starts declining with a minor exception at 4 expansions per iteration. Before analyzing the issue of optimal number of expansions per iteration further, the behavior (concave down) of the real time curve is analyzed.

Ignoring minor deviations, communication time and time taken for each expansion at a specified number of slaves remains the same for all iterations. Speedup decreases because as the number of expansions per iteration increases, the frequency with which slaves share information decreases. In other words, if slaves do more expansions before they return the information to the master, the delay between the information updates is greater. If the delay is too large, many of the expansions will be unnecessary, causing a decrease in speedup. On the other hand if this delay is too small, then the overhead involved in communication plays too large a role, again causing a decrease in speedup. Thus there is an optimal number of expansions per iteration which is the optimal value for E.

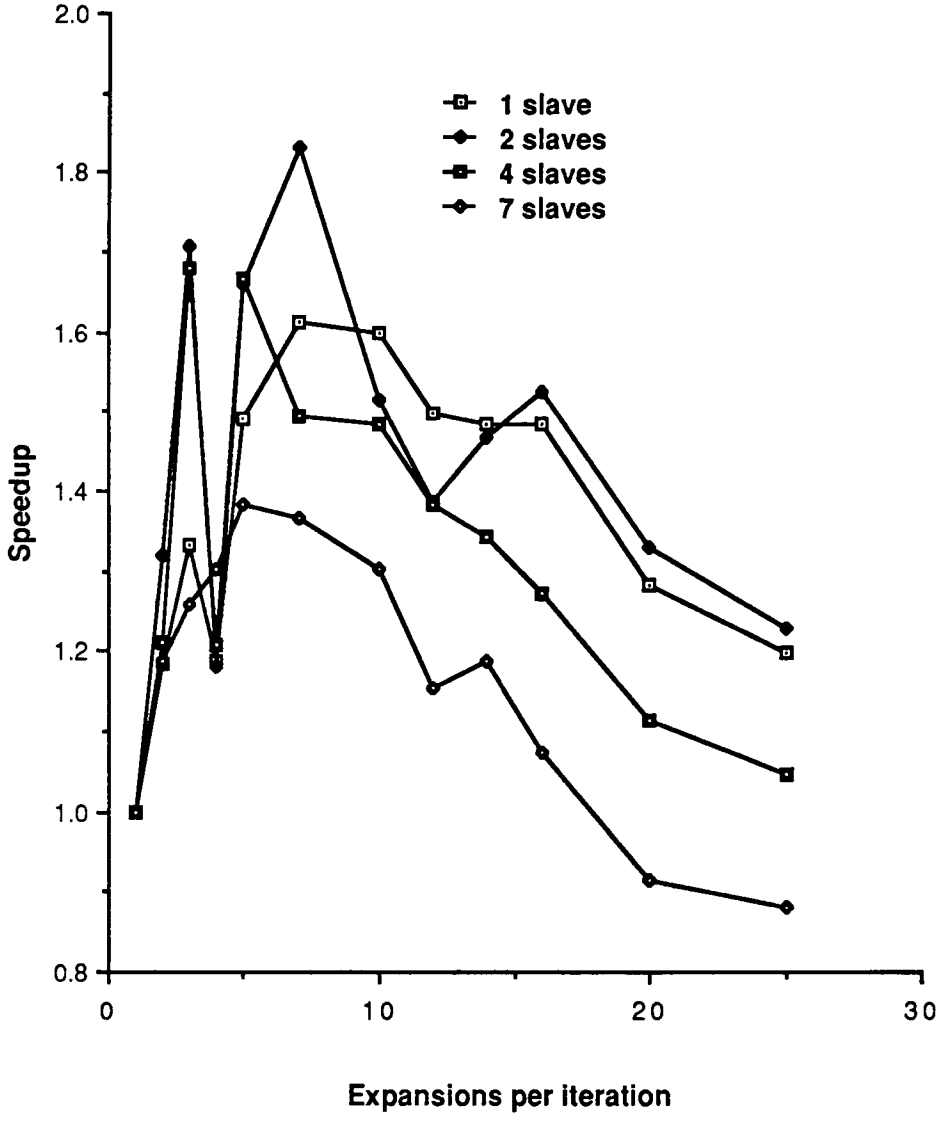

Figure 2. Real time speedup as a function of number of expansions per iteration.
Heuristic: Minimum spanning tree.

4.2 c_1-c_2 Time Versus Number of Expansions

The time to solve a problem may vary from run to run depending upon the load on the machine. Even though most of the experiments are conducted during hours when the lab is closed, it is always possible to remote login to the machines, since they are network connected. There may also be network problems every once in a while. This increases the time taken to solve a problem sometimes. Thus leading to bad real times (This is not to suggest that these problems occur all the time. Actually, these conditions/problems affected very few times for our experimental results). This section presents an alternative measure for computation time called c_1-c_2 time.

This leads to developing a prediction for elapsed time from expansion time (c_1) and communication time between the slaves and the master (c_2) in terms of time, number of iterations, and the number of expansions per iteration. The next step is to select the model. The simplest is to assume the following relation between time and the other quantities for a single problem.

$$T = I(E) * [E * c_1 + c_2] \qquad (4.1)$$

where

T = Real time in seconds.

$I(E)$ = number of iterations as a function of E.

E = Expansions per iteration.

c_1 = Time in seconds for each expansion.

c_2 = Time in seconds for communication between the master and slave.

c_1-c_2 time is the predicted elapsed time obtained from c_1, c_2, number of iterations and number of expansions per iteration. This measure of predicted elapsed time removes the inconsistencies that might occur in the real time measure due to some unusual conditions. Since c_1-c_2 time uses the number of iterations, it would not be affected by the unusual changes of computing conditions for the following reasons. And also, c_1 and c_2 times are averaged over 2000 problems. Synchronization of sending nodes to the slaves by the master makes the algorithm deterministic in computing the number of iterations. The algorithm is said to be deterministic if it gives the same output everytime for the same input. If the algorithm is asynchronous, using the number of iterations as a measure to compute the time is a bad choice.

Whenever the master sends a new set of nodes to the slaves for expansion, it is considered as one iteration. The time taken on the slaves per each iteration is the number of expansions times the time for each expansion (c_1). C_1 and c_2 are important because the optimal number of expansions depend on these two factors. If the communication time is high, there should be more expansions per iteration or vice versa. Otherwise, communication cost compensates for the gains made by doing frequent updates. This can only be achieved by controlling the number of expansions per iteration, which is under the software control.

The model for c_1-c_2 time gives a "time" value for any given values for c_1 and c_2. We wish to present graphs for the c_1 and c_2 appropriate for the actual

experiments. To do so, we could measure c_1 and c_2 directly, or estimate them from our data. We choose to do the latter using a least squares technique for 2 variables. Let

$$F(c_1,c_2) = \sum_{i=1}^{N} [T_i - (I_i E_i * c_1 + I_i c_2)]^2 = 0$$

where

N = number of problems,

and T_i , I_i , and E_i are data collected from the i th experiment. Thus F is the sum of the squares of the differences between the measured time (T) and the predicted c_1-c_2 time.

To minimize this difference, the two equations,

$$\frac{dF}{dc_1}(c_1,c_2) = 0 \quad \text{and} \quad \frac{dF}{dc_2}(c_1,c_2) = 0$$

must be solved simultaneously for the values of c_1 and c_2.

This leads to (see appendix A for details of math),

$$c_1 = \frac{\sum (T_i I_i E_i) \sum I_i^2 - \sum (T_i I_i) \sum I_i^2 E_i}{\sum (I_i E_i)^2 \sum I_i^2 - (\sum I_i^2 E_i)^2}$$

$$c_2 = \frac{\sum (T_i I_i E_i) \sum I_i^2 E_i - \sum (T_i E_i) \sum (I_i E_i)^2}{(\sum I_i^2 E_i)^2 - \sum (I_i E_i)^2 \sum I_i^2}$$

The parameter number of slaves is not introduced in the c_1, c_2 formula to keep the model simple. Ignoring the number of slaves should not be a great concern, since the emphasis is to find how the values of c_1 and c_2 affect the software controlled parameter, the number of expansions per iteration. Before going to the next set of graphs using c_1 and c_2, the effect of the number of iterations on the speedup is briefly discussed since iterations are an important factor in c_1-c_2 time.

For most problem instances, the number of iterations should decrease or stay the same as the number of expansions per iteration increases. The reason is as E increases, the slaves can send more information to the master on each iteration. This normally results in better or the same pruning of the nodes. As a consequence, the number of sets of problems sent to the slave (i.e., number of iterations) decreases or remains the same. The decrease in the number of iterations is not significant beyond a certain number of expansions. The number of iterations may even stay constant. The reason for this behavior is the following. If the number of expansions per iteration is greater, then it might lead to more unnecessary expansions. This is because of the cost of making these unnecessary expansions may outweigh the gains made by the additional information generated by these expansions. This reduces the speedup. However, this does not increase the

number of iterations since the algorithm is synchronized. Thus, slaves take more time per iteration if E is greater. This decreases the speedup if there is no corresponding decrease in the number of iterations.

The values of c_1 and c_2 are computed by doing a least squares minimization over 2000 problem instances. The results are presented in the table 4.1 for the minimum spanning tree heuristic.

Table 4.1. Values of c_1 and c_2 for minimum spanning tree heuristic.

Number of slaves	c_1 (Seconds)	c_2 (Seconds)
1	0.006135	0.190052
2	0.009011	0.225764
4	0.014969	0.261777
7	0.029277	0.306797

As the number of slaves increases, both c_1 and c_2 increase. The reason for the increase in c_1 may be the following. Once the slaves are finished with the expansions for a given iteration, sometimes they may have to wait a while before sending the data. This is because the communication channel to the master may be busy since one of the other slaves may be using it to send its information. The waiting period may increase as the number of slaves increases. The increase in c_2 may be due to the fact that the master has to receive more information as the

number of slaves increases. Several issues are discussed in the following graphs that are developed using c_1-c_2 time.

Figure 3 presents c_1-c_2 time versus the number of expansions per iteration. The time here is equal to the time as defined in equation 4.1 with c_1 and c_2 set as obtained from the least-squares minimization (Table 4.1). I(E) is taken from the data. The heuristic used is minimum spanning tree (MST). Each data point is an average over 100 random problem instances. As expected, time decreases initially and increases again as the number of expansions per iteration (E) increases. Figure 4 presents the speedup curve for this time vs E at different number of slaves. Here the speedup at e expansions per iteration is defined as the ratio of c_1-c_2 time when the number of expansions per iterations is one, divided by the c_1-c_2 time when the number of expansions per iteration is e. As can be seen from the graph, the optimal number of expansions varies depending on the number of slaves. In figures 3 and 4, the optimal number is 14 for one slave, 7 for 2 slaves, 5 for 4 slaves and 3 for 7 slaves. Thus, the optimal number decreases as the number of slaves increases. This is reasonable since as the number of slaves increases, the combined information generated by the slaves for each expansion is greater. Thus, there is less to be gained by further increasing the amount of information by increasing the number of expansions per iteration. As pointed out earlier, there is more delay in sharing this information if the number of expansions per iteration is high. This is a concern if the number of slaves is high . This causes a decrease in the optimal number of expansions as the number of slaves increases. The increase in speedup at the optimal point also decreases as the number of slaves increases. In the case of one slave, there is almost a 40%

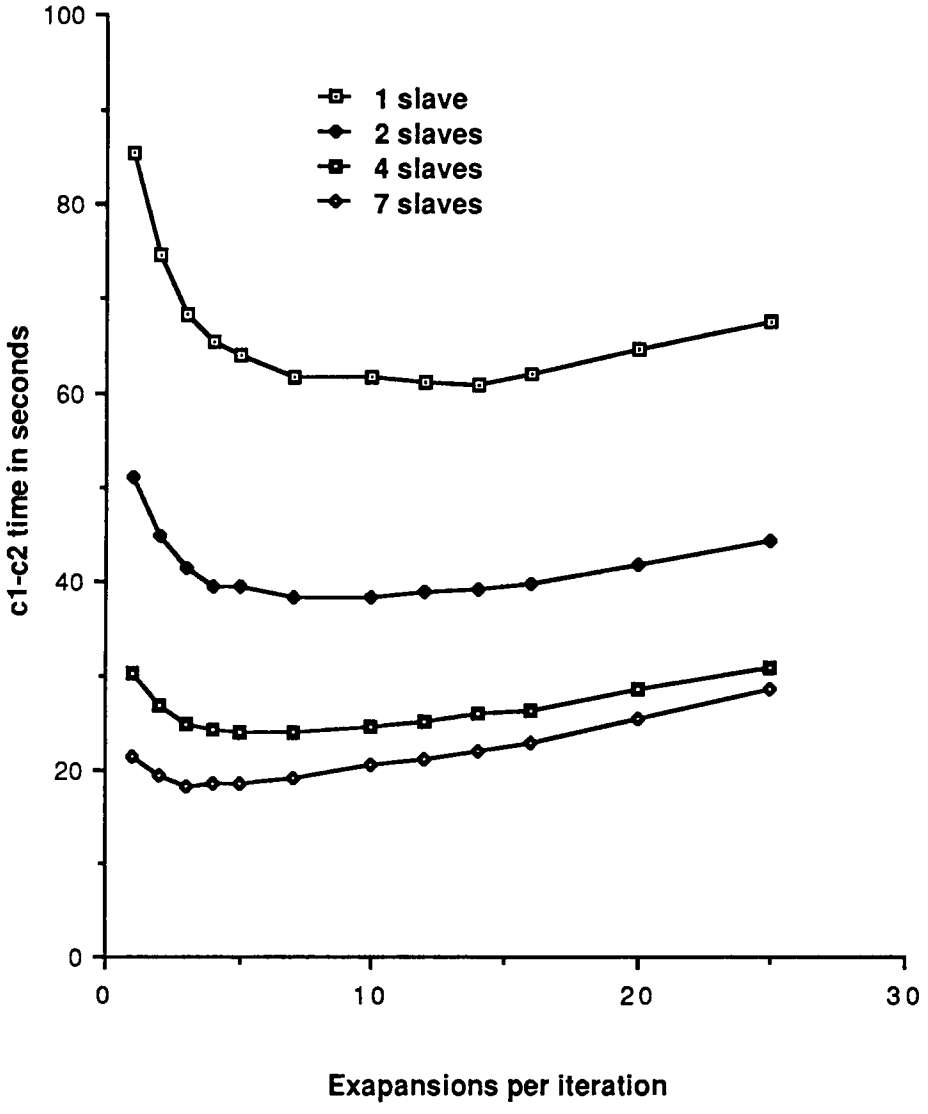

Figure 3. Average c_1-c_2 time as a function of number of expansions per iteration.
Heuristic: Minimum spanning tree.

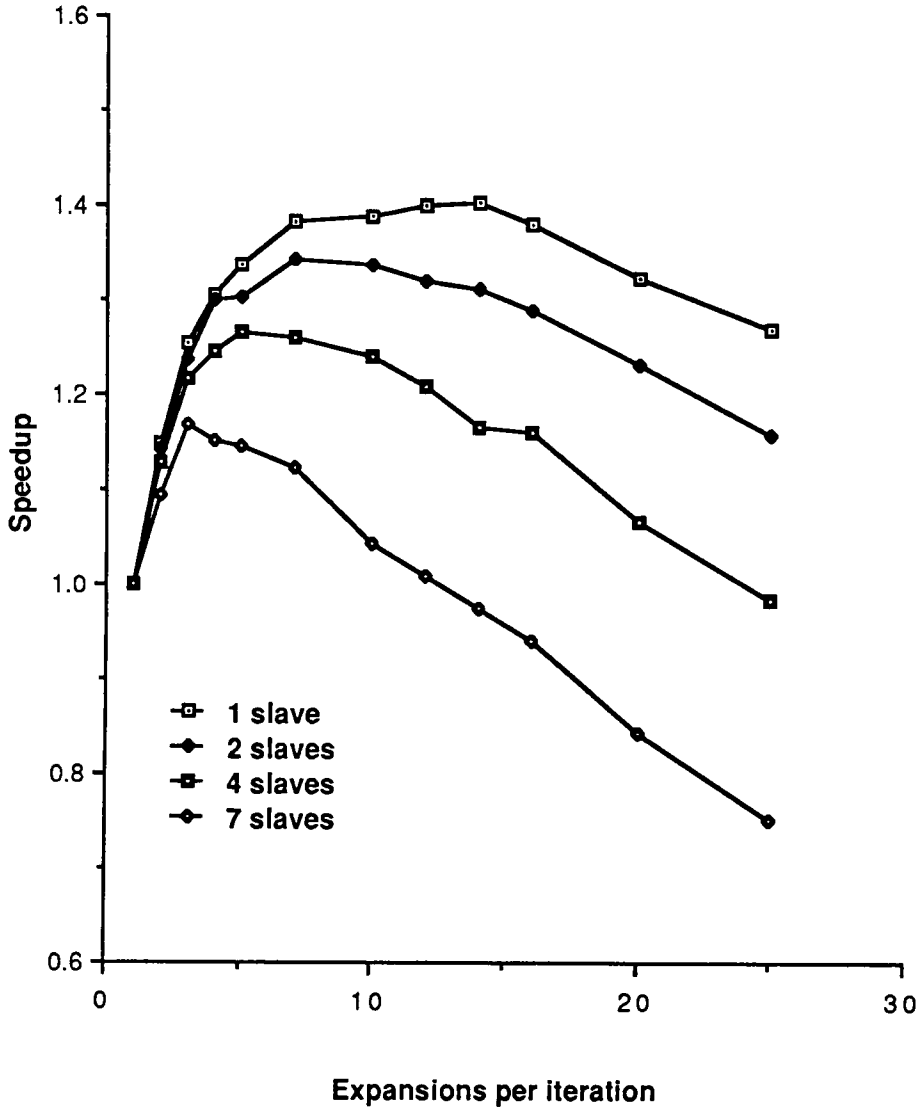

Figure 4. c_1-c_2 time speedup as a function of number of expansions per iteration.
Heuristic: Minimum spanning tree.

speedup at the optimal point. This speedup is 32%, 25%, and 18% for 2, 4, and 7 slaves, respectively.

By varying c_2, we can estimate c_1-c_2 time on networks with communication times different from the network actually used for the experiments. Figure 5 presents improvement due to an optimal choice for E at different communication times for different numbers of slaves. The factor of c_2 is varied from 0.1 to 10.0. Factor 0.1 means that the communication time (c_2) is slashed to one tenth of that of figure 3 or table 4.1. For each c_2, the ratio of time taken at one expansion to the least time taken at any expansion (optimal number of expansions) at that c_2 is taken as improvement. The number of expansions per iteration varies from 1 to 25. This graph demonstrates the importance of the optimal number of expansions and how it varies at different communication times. The graph shows that the improvement achieved for different c_2's is considerable. This is especially true for higher communication costs. Initially at a 0.1 factor, the improvement is negligible but as the factor increases to 10.0, it becomes very important. For example, in case of 2 slaves at a factor of 1.0, the improvement is 25% but when the factor is 10.0, the improvement (1exp/E) is 100%. In other words, in a network whose communication time is ten times faster than that of the network we used, we predict that the total elapsed time at the optimal number of expansions is one fourth of what we measured in our network. This is true for all number of slaves. If c_2 is too low relative to c_1, it does not cost much to communicate after each expansion. If c_2 is too high relative to c_1, it becomes too expensive to communicate frequently to the master. This is quite evident from

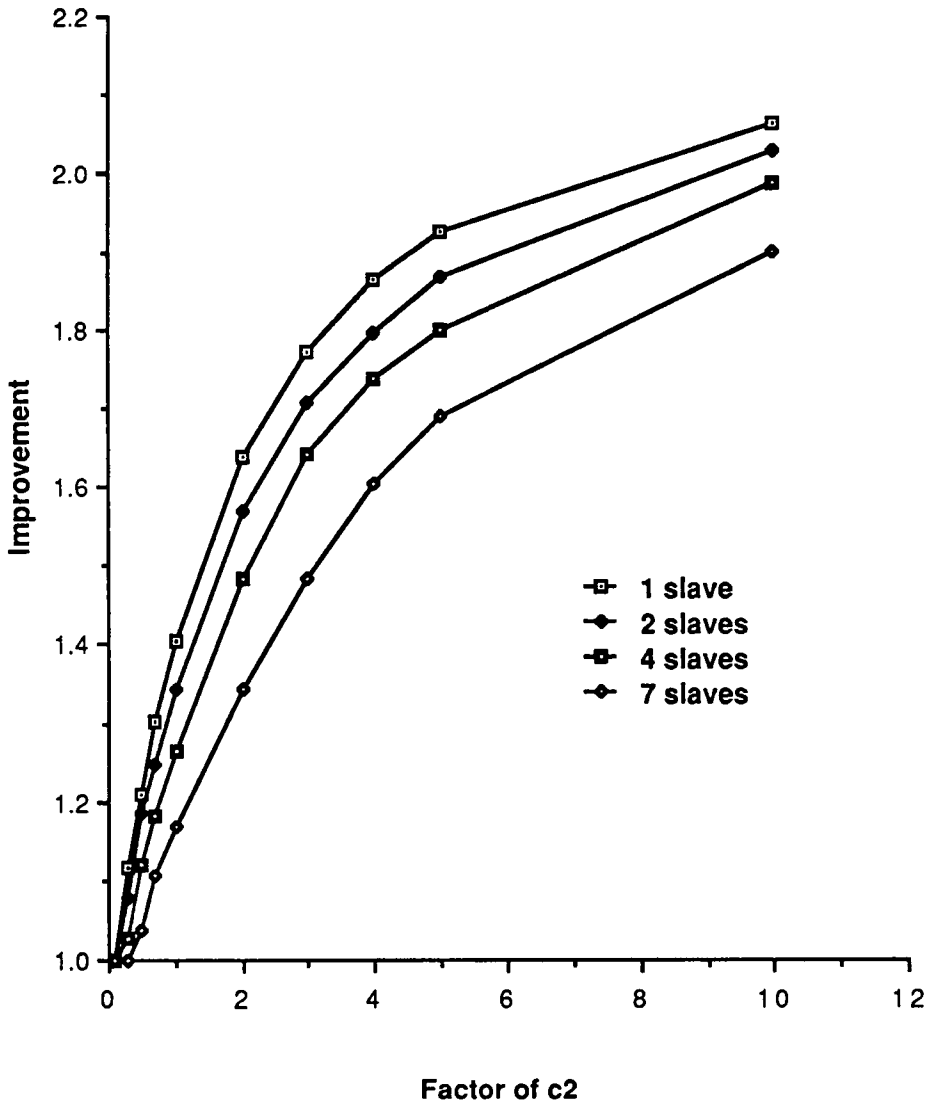

Figure 5. c_1-c_2 time speedup as a function of factor of c_2. c_2 values at factor 1.0 are from table 4.1.
Heuristic: Minimum spanning tree.

figure 5. It also shows that granylarity (E) is a function of compute-to-communicate ratio.

Figure 6 presents the speedup, based on c_1-c_2 time, at different values for E (expansions per iteration) for several communication costs. The number of slaves is 1 and speedup is defined as before. The heuristic used is minimum spanning tree (MST). The objective is to demonstrate the effect of communication cost on the optimal number of expansions and its importance especially at higher communication costs. When the communication cost is factorized with 0.1, the speedup starts falling after one expansion. That is, increasing E degrades performance of the algorithm. But at factors 4 and 10, even when E is 25 the speedup is still going upwards. Both trends are expected because of their respective communication costs. For other factors of c_2, the speedup initially goes up and then decreases. As the communication cost increases, the optimal point steadily rises. This trend is very consistent with the analysis of previous graphs.

It is worth noting a point about the single slave. When there is only one slave, what is the necessity of sending information to the master since the master is not doing any expansions? The reason is this. Even though the master is not doing any expansions, it controls the open list with the list of the nodes to be expanded. The master sends only one node (best node) to the slave in each iteration and receives the information and updates its lists. So, it is necessary to integrate the information that the slave sends into the information that the master has to select the best node. This necessitates the slave to communicate with the master.

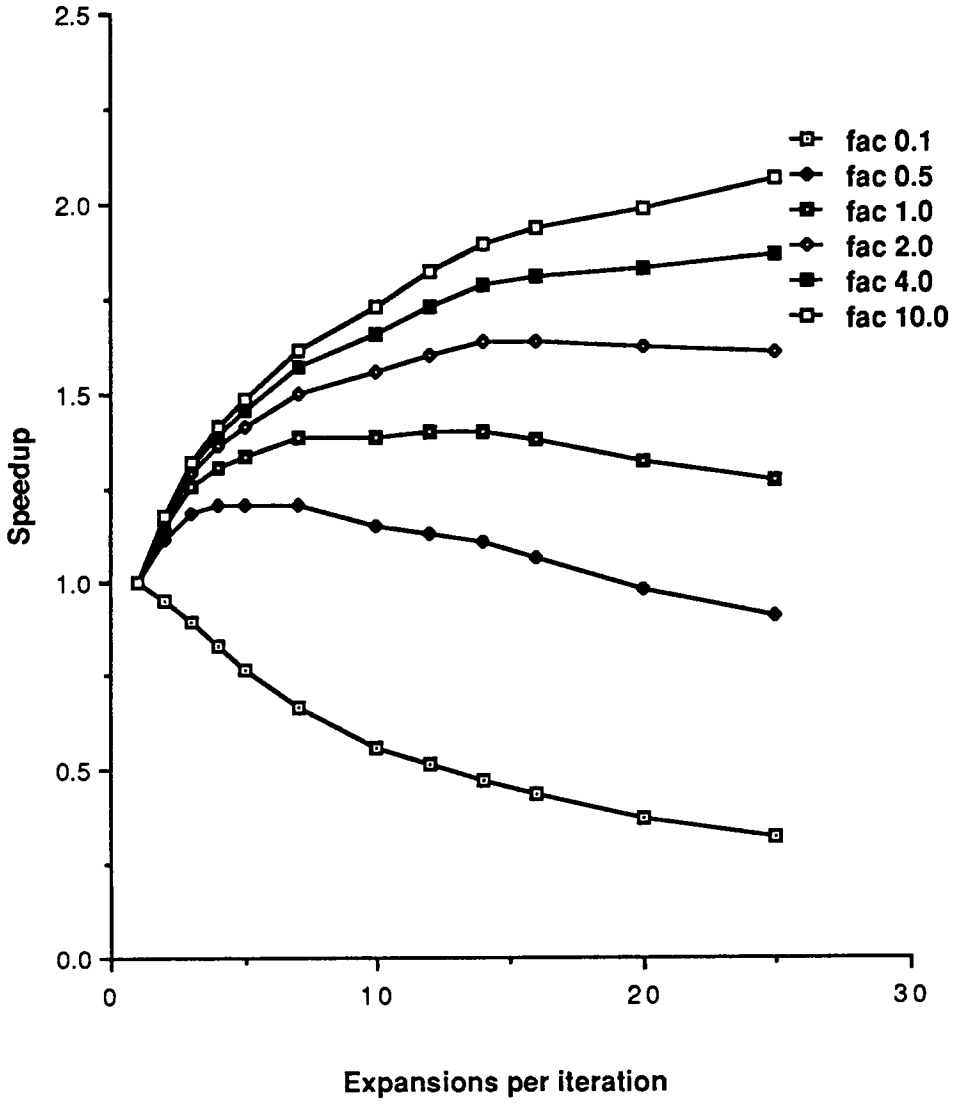

Figure 6. c_1-c_2 time speedup as a function of number of expansions per iteration at different factors of c_2. c_2 values at 1.0 are from table 4.1. Number of slaves: 1
Heuristic: Minimum spanning tree.

The same trend as in figure 6 is observed in figures 7, 8, and 9 which contain the data for 2, 4, and 7 slaves, respectively. The increase in speedup decreases as the number of slaves increases. The reason for this may be due to increase in overhead with the increase in number of slaves. This overhead involves creating slaves and increased information that needs to be sorted by the master. This tendency of lower speedup can be seen by comparing figure 6 with figure 9. The speedup is 110% when the factor is 10.0 for one slave in figure 6 and the speedup is only about 80% for 7 slaves at the same factor.

Figure 10 presents the effect of varied expansion times (c_1) on speedup at different expansions per iteration. The number of slaves are one and the heuristic is minimum spanning tree. This graph demonstrates the importance of the optimal number of expansions especially at low expansion times (c_1). As the time for each expansion increases, the speedup decreases rapidly with the increasing number of expansions. In other words, the optimal number of expansions decreases with the increase in time for each expansion. The speedup is actually negative when the factors are 4.0 and 10.0 as the number of expansions per iteration increases. The increase in optimal number of expansions with the increase in c_1 is due to the following. If c_1 is much higher relative to c_1, it is not cost effective to make too many expansions per iteration (i.e., before updating the information). The effect of this communication-to-computation ratio is similar to the trend observed in figures 6 to 9. The same trend can also be observed in figures 11, 12, and 13 when the number of slaves are 2, 4, and 7, respectively.

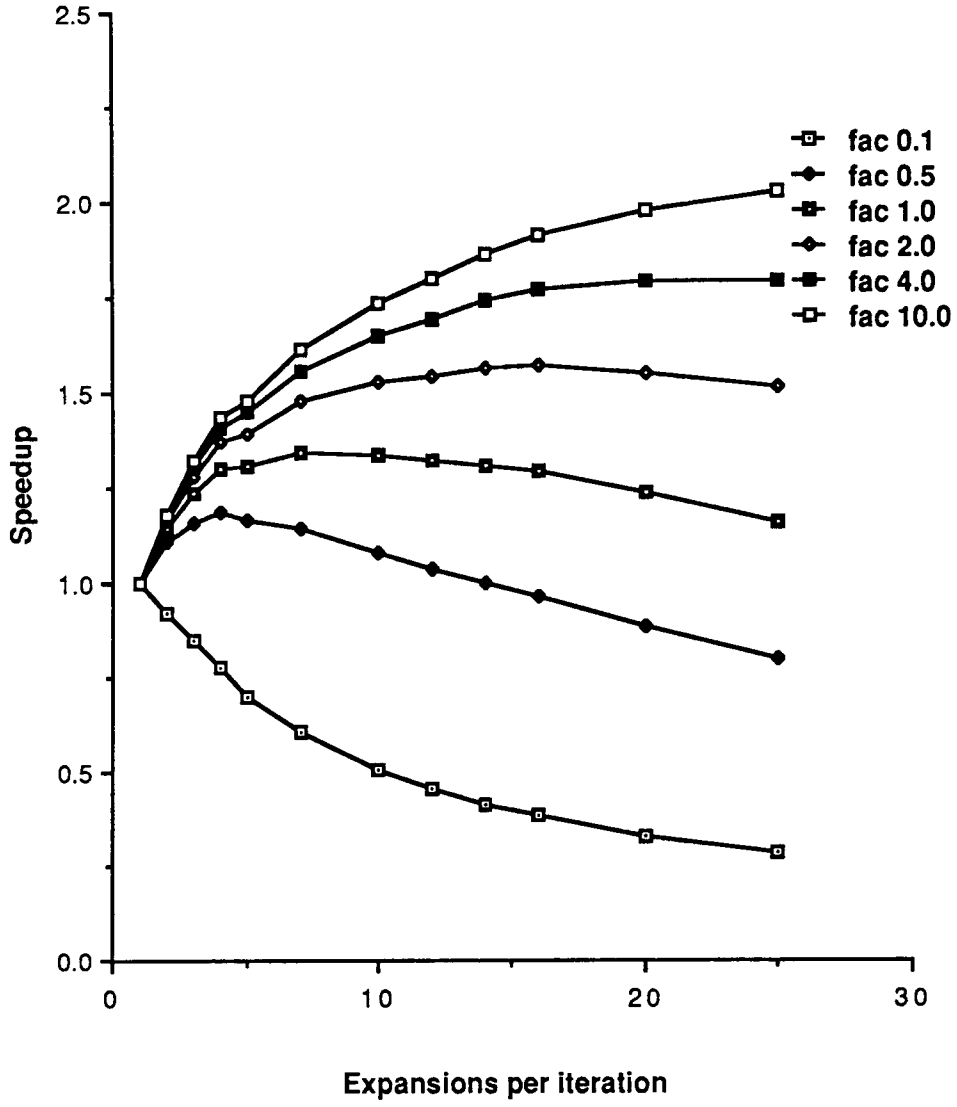

Figure 7. c_1-c_2 time speedup as a function of number of expansions per iteration at different factors of c_2. c_2 values at 1.0 are from table 4.1. Number of slaves: 2
Heuristic: Minimum spanning tree.

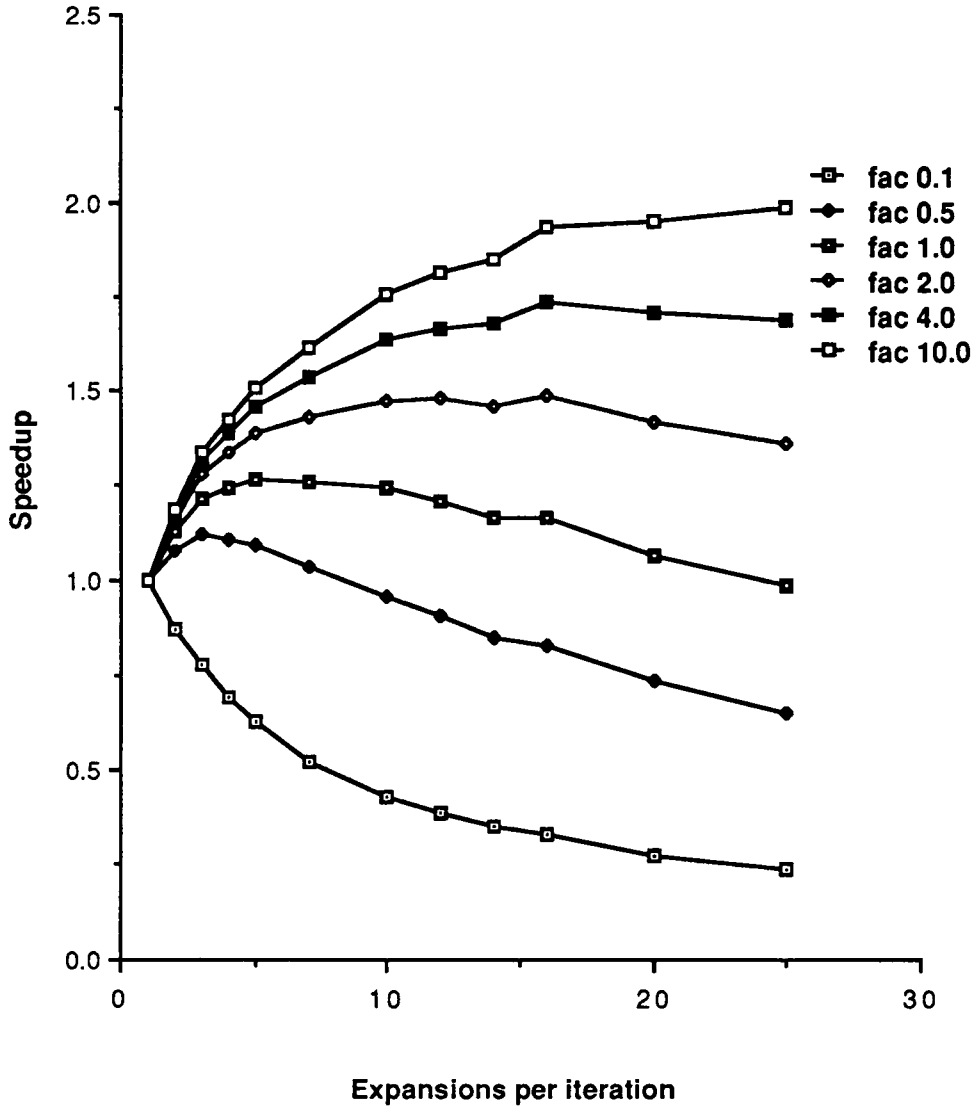

Figure 8. c_1-c_2 time speedup as a function of number of expansions per iteration at different factors of c_2. c_2 values at 1.0 are from table 4.1. Number of slaves: 4
Heuristic: Minimum spanning tree.

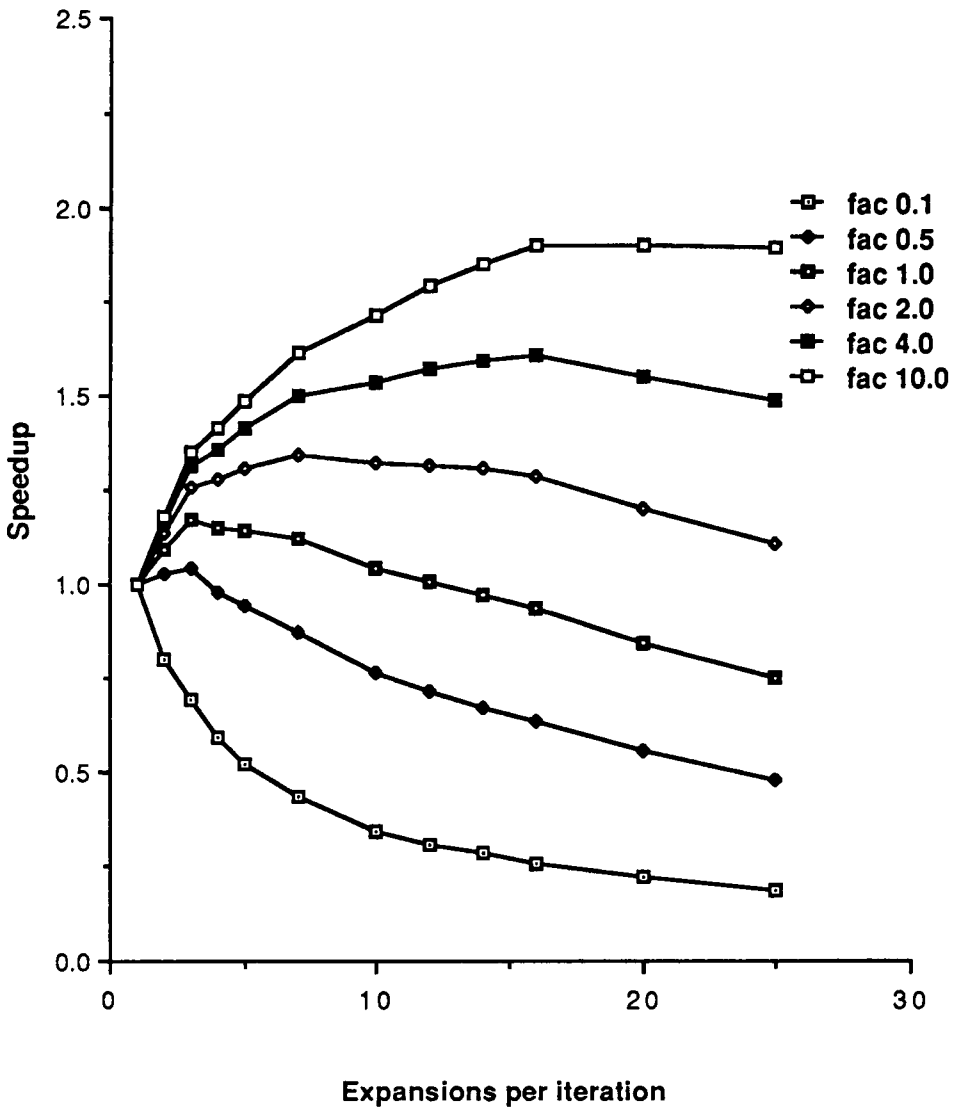

Figure 9. c_1-c_2 time speedup as a function of number of expansions per iteration at different factors of c_2. c_2 values at 1.0 are from table 4.1. Number of slaves: 7
Heuristic: Minimum spanning tree.

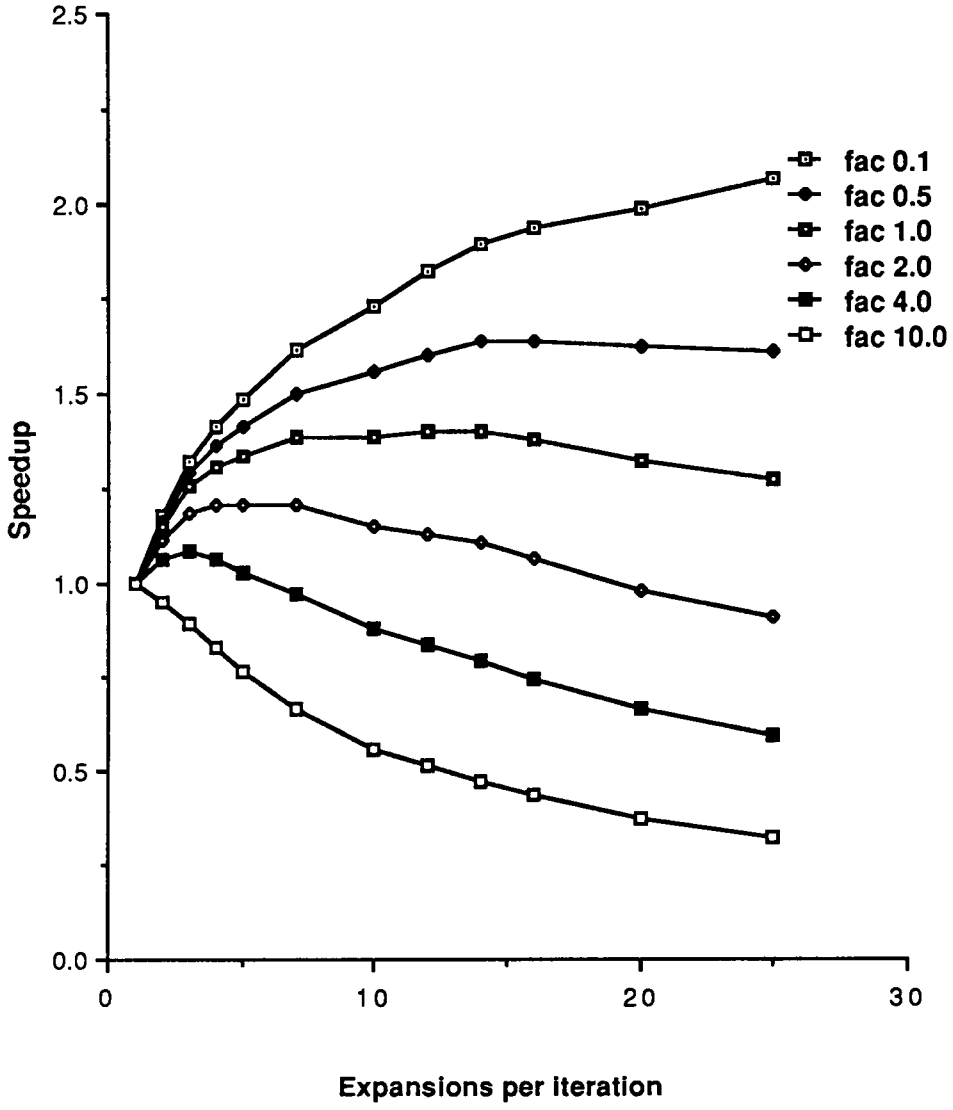

Figure 10. c_1-c_2 time speedup as a function of number of expansions per iteration at different factors of c_1. c_1 values at 1.0 are from table 4.1. Number of slaves: 1 Heuristic: Minimum spanning tree.

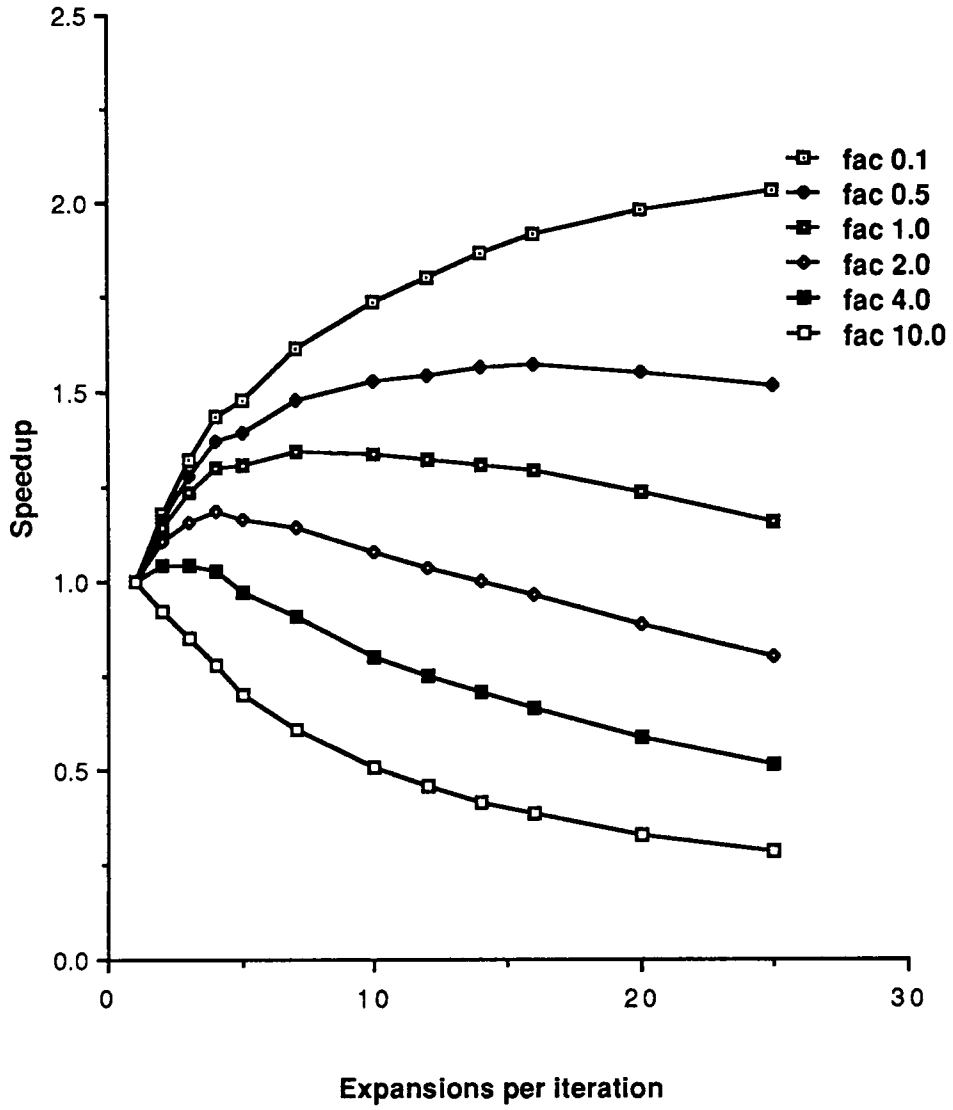

Figure 11. c_1-c_2 time speedup as a function of number of expansions per iteration at different factors of c_1. c_1 values at 1.0 are from table 4.1. Number of slaves: 2 Heuristic: Minimum spanning tree.

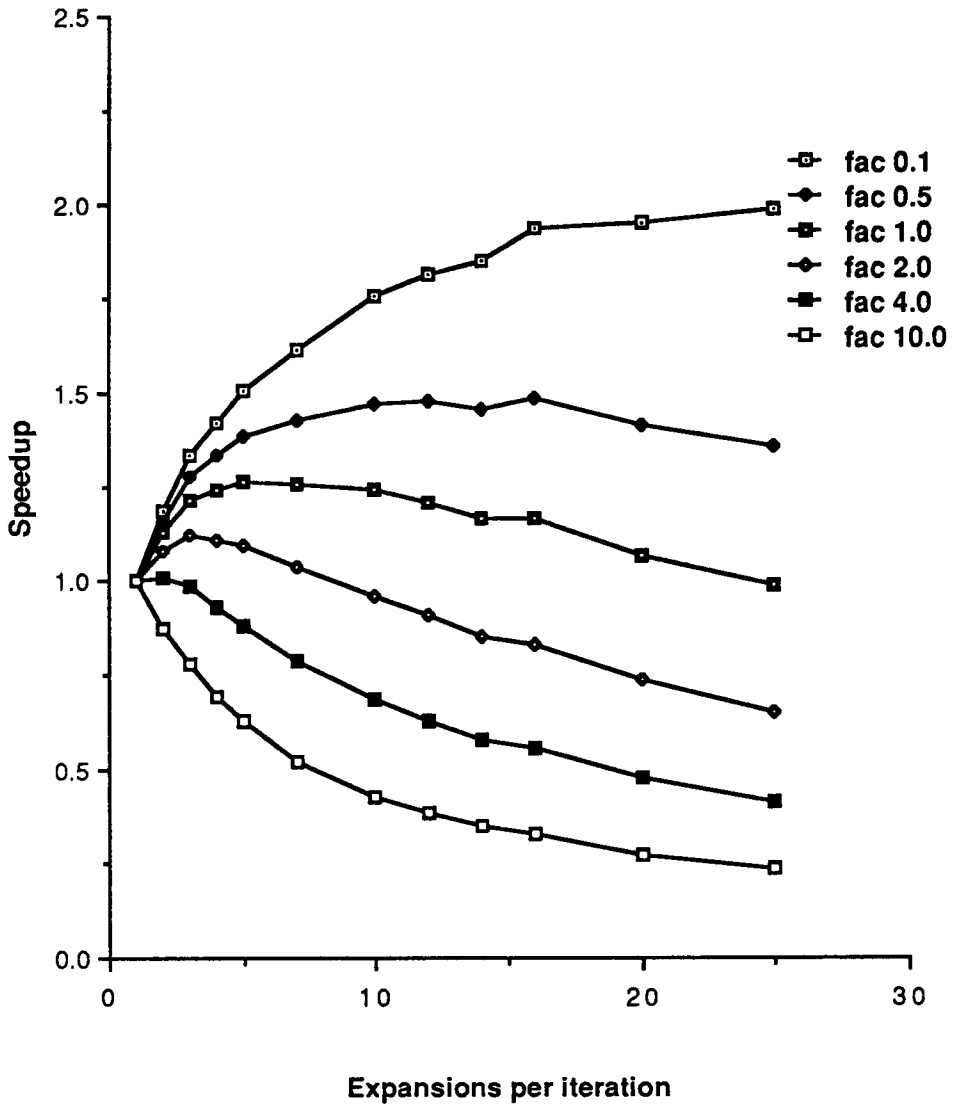

Figure 12. c_1-c_2 time speedup as a function of number of expansions per iteration at different factors of c_1. c_1 values at 1.0 are from table 4.1. Number of slaves: 4 Heuristic: Minimum spanning tree.

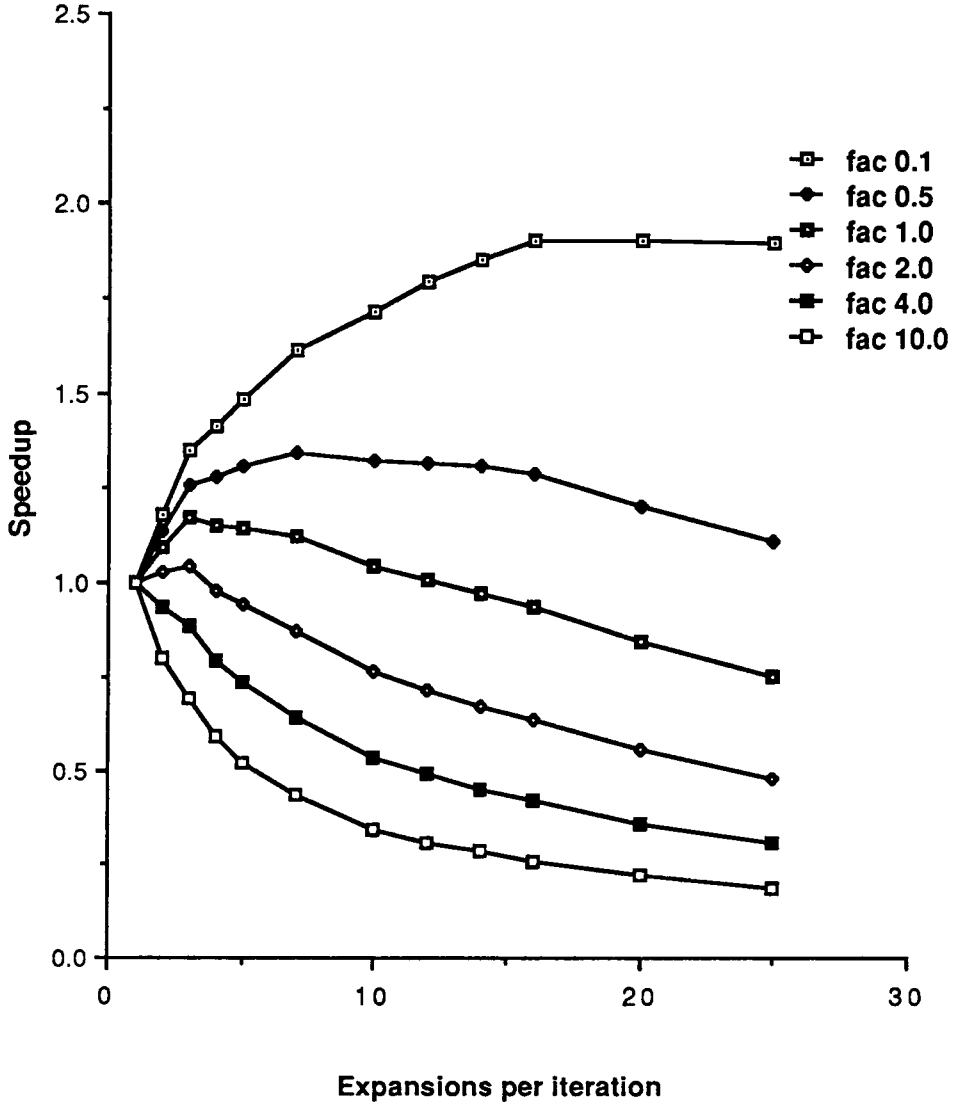

Figure 13. c_1-c_2 time speedup as a function of number of expansions per iteration at different factors of c_1. c_1 values at 1.0 are from table 4.1. Number of slaves: 7 Heuristic: Minimum spanning tree.

Figures 5 to 13 clearly demonstrate the positive and adverse effects of varying expansions per iteration at various communication costs. The optimal value for E, at which higher speedups are achieved, is found to be a crucial factor for better speedups. It is also evident that increasing the number of slaves causes the optimal value for E to decrease.

4.3 Effect of the Heuristic

About 2000 problems are solved using two different heuristics. The results are compared to see the effects on speedup and other parameters. The results vary quite considerably from minimum-spanning-tree heuristic to the nearest-neighbor heuristic. This comparison of heuristics is to demonstrate the effect of heuristics on different problem parameters. This is crucial since heuristics guide the search process. As will be seen in the following discussion, the minimum spanning tree heuristic in general results in better speedup than the nearest neighbor.

Figure 14 presents real time (measured) versus the number of expansions per iteration (E). E varies from 1 to 7, the number of slaves from 1 to 7 and the heuristic is nearest neighbor. The time taken is very high compared to figure 1 which uses the minimum spanning tree heuristic. For example, the time taken for 2 slaves at E = 3 is about 330 seconds in the case of nearest neighbor whereas the minimum spanning tree (MST) heuristic takes about 35 seconds. This result is consistent for different number of slaves at all values for E. The reason for this

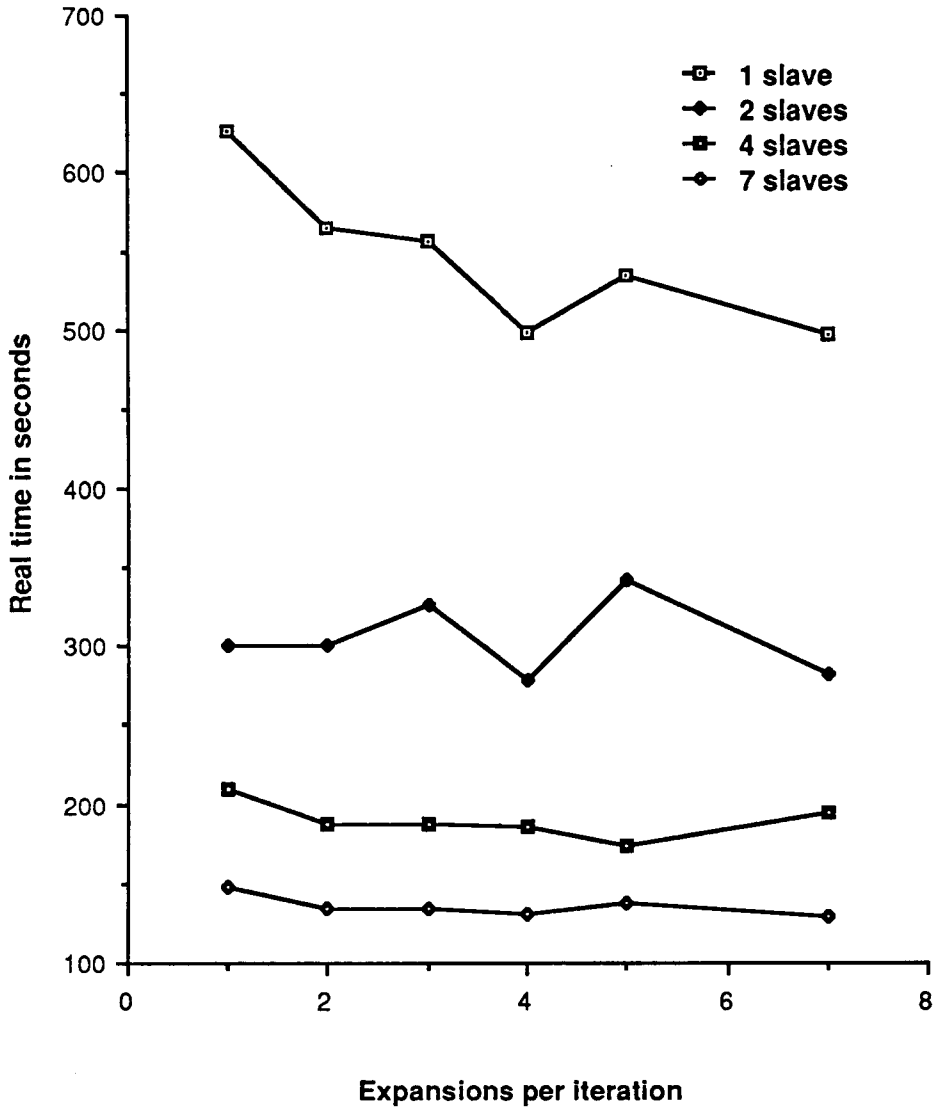

Figure 14. Average real time as a function of number of expansions per iteration.
Heuristic: Nearest neighbor.

could be attributed to the fact that in computing the heuristic value, minimum spanning tree builds a complete tree from all the unexpanded nodes for that state. It then estimates the minimum possible cost to reach the goal node. In other words, MST returns the minimum possible value to connect all the unexpanded nodes of a given state as a heuristic value. But nearest neighbor just returns the cost from the current node to its nearest unexpanded neighbor node. Thus, the value returned by MST is always higher or the same as that of nearest neighbor. It is usually the same when the solution is just one arc away. In almost all other times, MST returns a strictly larger value than the nearest neighbor. However, both the heuristics underestimate the real value. If the underestimated value is higher, it tends to be closer to the actual value. Also, the higher value in the case of MST makes the pruning mechanism more efficient [2]. This results in fewer states and fewer expansions. The consequences of this are higher speedup and fewer iterations. Experimental results shown in figure 14 are consistent with this analysis.

Figure 15 presents the speedup versus the number of expansions per iteration (E) for 1 and 7 slaves and for both the nearest neighbor and MST heuristics. The time used is the real measured time. E varies from 1 to 7. Speedup is higher in the case of minimum spanning tree (MST) at every value for E with the exception of 4. The reasons for lower speedup at 4 is the same as discussed in section 1. Thus, the minimum-spanning heuristic has a clear superiority over the nearest-neighbor heuristic in both real time and speedup. The optimal number of expansions is higher in the case of nearest neighbor. For example, in the case of MST for 7 slaves, the optimal value is 3 where as in the

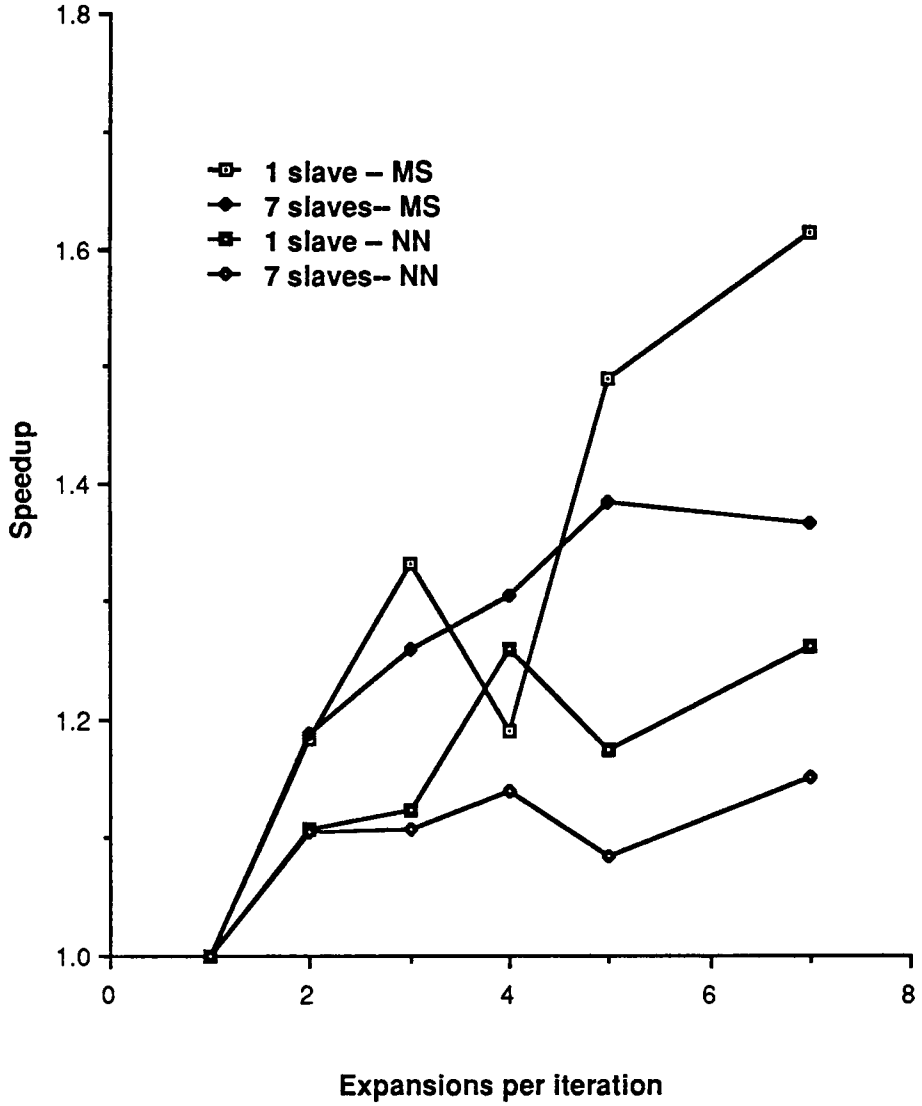

Figure 15. real time speedup as a function of number of expansions per iteration.
Heuristics: Minimum spanning tree(MS) and Nearest neighbor (NN).

case of nearest neighbor, the speedup is still rising after 7. This is expected since c_1 for nearest neighbor is much less than that of MST but communication time (c_2) is almost the same. This result is quite consistent with the earlier discussion.

The time for each expansion (c_1) and communication time (c_2) are computed for nearest neighbor heuristic, using a least squares estimate as for the MST heuristic. Results for 1 and 7 slaves are presented in table 4.2.

Table 4.2. Values of c_1 and c_2 for nearest neighbor heuristic.

Number of slaves	c_1 (Seconds)	c_2 (Seconds)
1	0.002482	0.211755
7	0.010215	0.347871

Communication time is almost the same in the cases of both MST and nearest neighbor. The volume of data is same in both the cases. However, the expansion time of MST is three times higher than the nearest neighbor. This is quite logical since MST has to build a tree to arrive to a heuristic value. The nearest neighbor heuristic just has to find the smallest cost arc from the current node to the nodes yet to be examined.

Figure 16 presents the speedup versus the number of expansions for 1 and 7 slaves but the time used is the c_1-c_2 predicted time. Similar to the case of

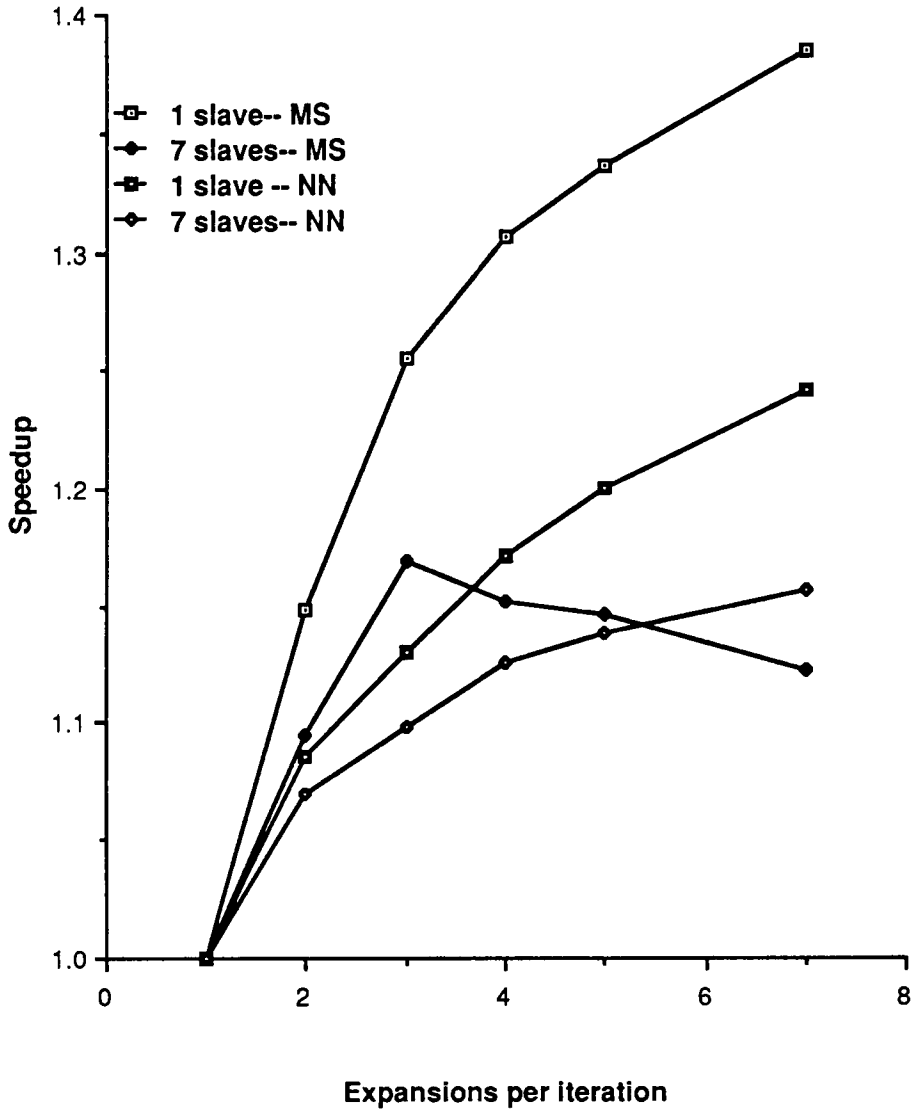

Figure 16. c_1-c_2 time speedup as a function of number of expansions per iteration.
Heuristics: Minimum spanning tree(MS) and Nearest neighbor (NN).

graph 15, speedup is higher in the case of minimum spanning tree for 1 and 7 slaves with the following exception. The speedup for MST is less than the nearest neighbor heuristic in the case of 7 slaves when E is greater than 5. However, even beyond E = 5, the real time is still less in case of MST than nearest neighbor.

Figures 17, 18, 19, and 20 present the effects of varying expansion time, c_1, on speedup in the case of nearest neighbor heuristic. The slaves are from 1 to 7 and E varies from 1 to 7. As c_1 increases, the optimal value for E decreases. The reasons are same as explained in the case of MST. However, the speedup is smaller compared to the MST heuristic data. Nevertheless, the result is the same in both the heuristics. Thus, it further justifies the importance of the optimal value depending on the values of c_1 and c_2.

The above discussion, however, points out the importance of having a powerful heuristic. Pruning power is more important than the speed of computing the heuristic. This is quite visible in the comparison of the two heuristics, nearest neighbor and minimum spanning tree.

4.4 Effect of Number of Slaves on Optimal Values

In this section, the effects of number of slaves on speedup and optimal points are presented. Figure 21 presents the average c_1-c_2 time to solve a set of problems at 1, 2, 4 and 7 slaves. The number of expansions per iteration (E)

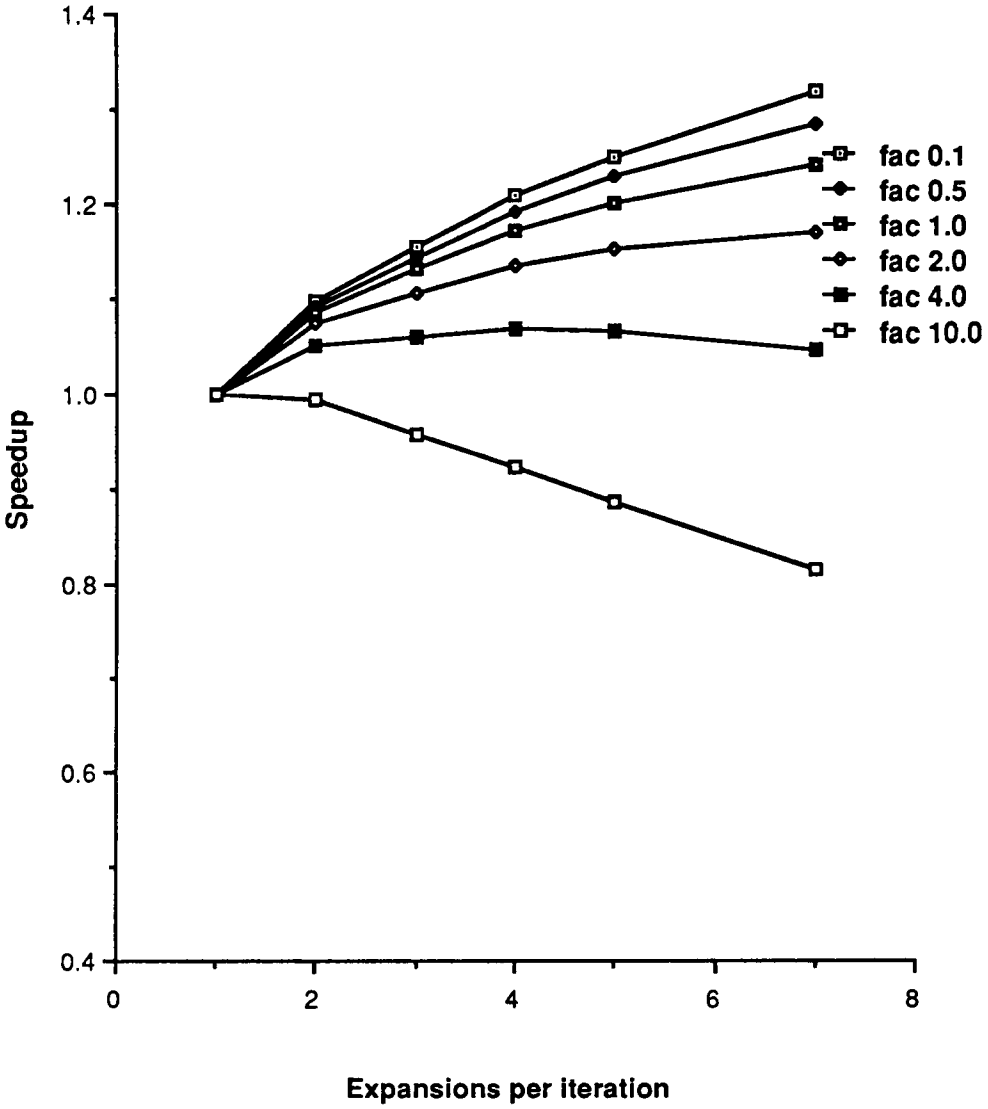

Figure 17. c_1-c_2 time speedup as a function of number of expansions per iteration at different factors of c_1. c_1 values at 1.0 are from table 4.2. Number of slaves: 1 Heuristic: Nearest neighbor.

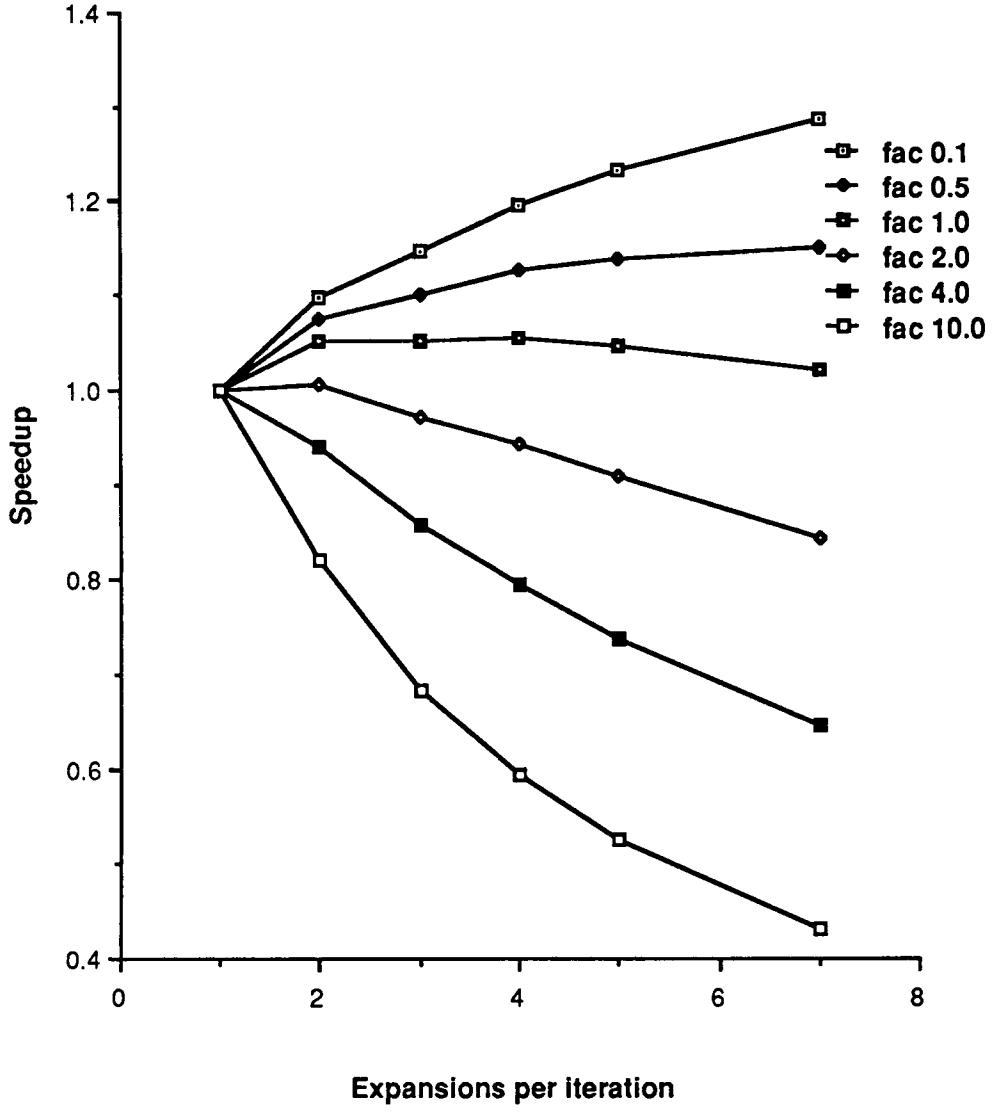

Figure 18. c_1-c_2 time speedup as a function of number of expansions per iteration at different factors of c_1. c_1 values at 1.0 are from table 4.2. Number of slaves: 2 Heuristic: Nearest neighbor.

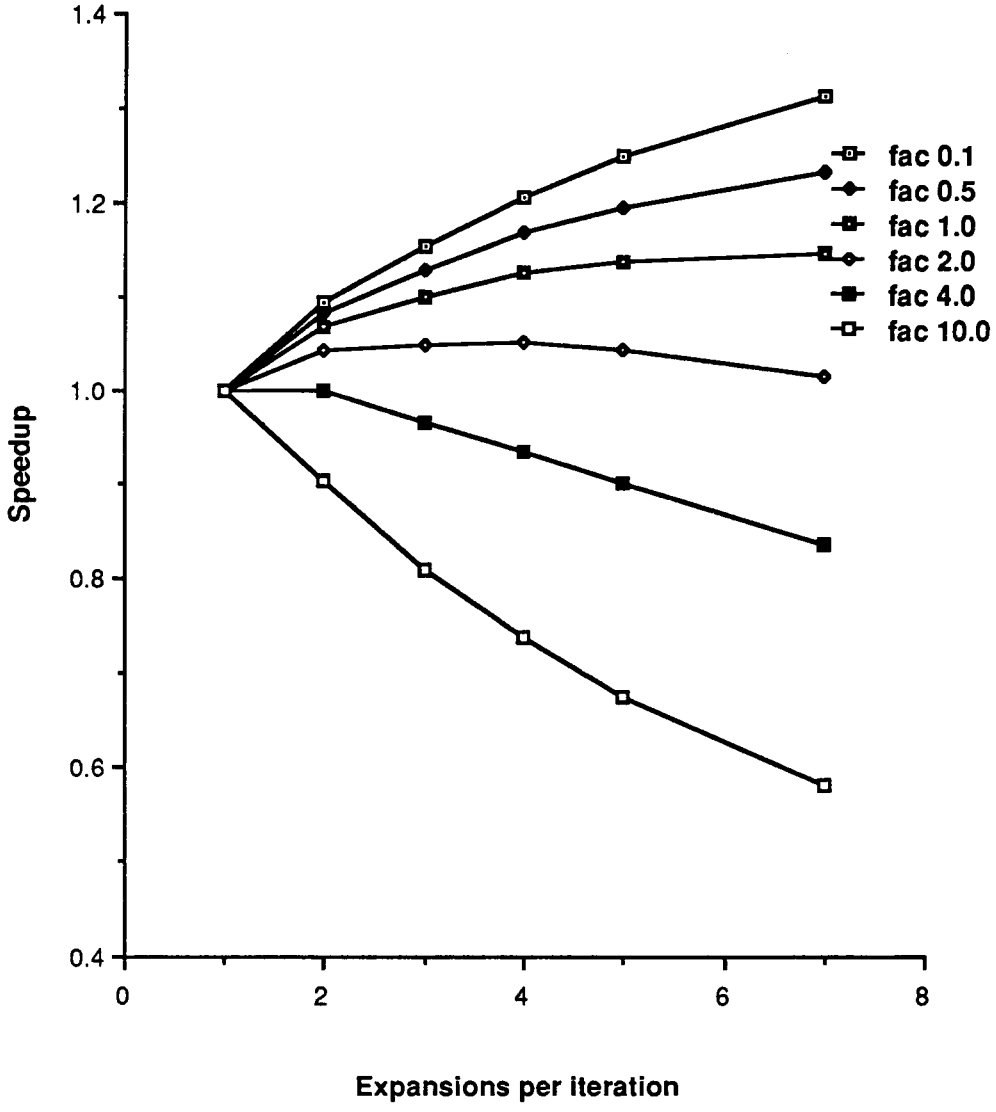

Figure 19. c_1-c_2 time speedup as a function of number of expansions per iteration at different factors of c_1. c_1 values at 1.0 are from table 4.2. Number of slaves: 4 Heuristic: Nearest neighbor.

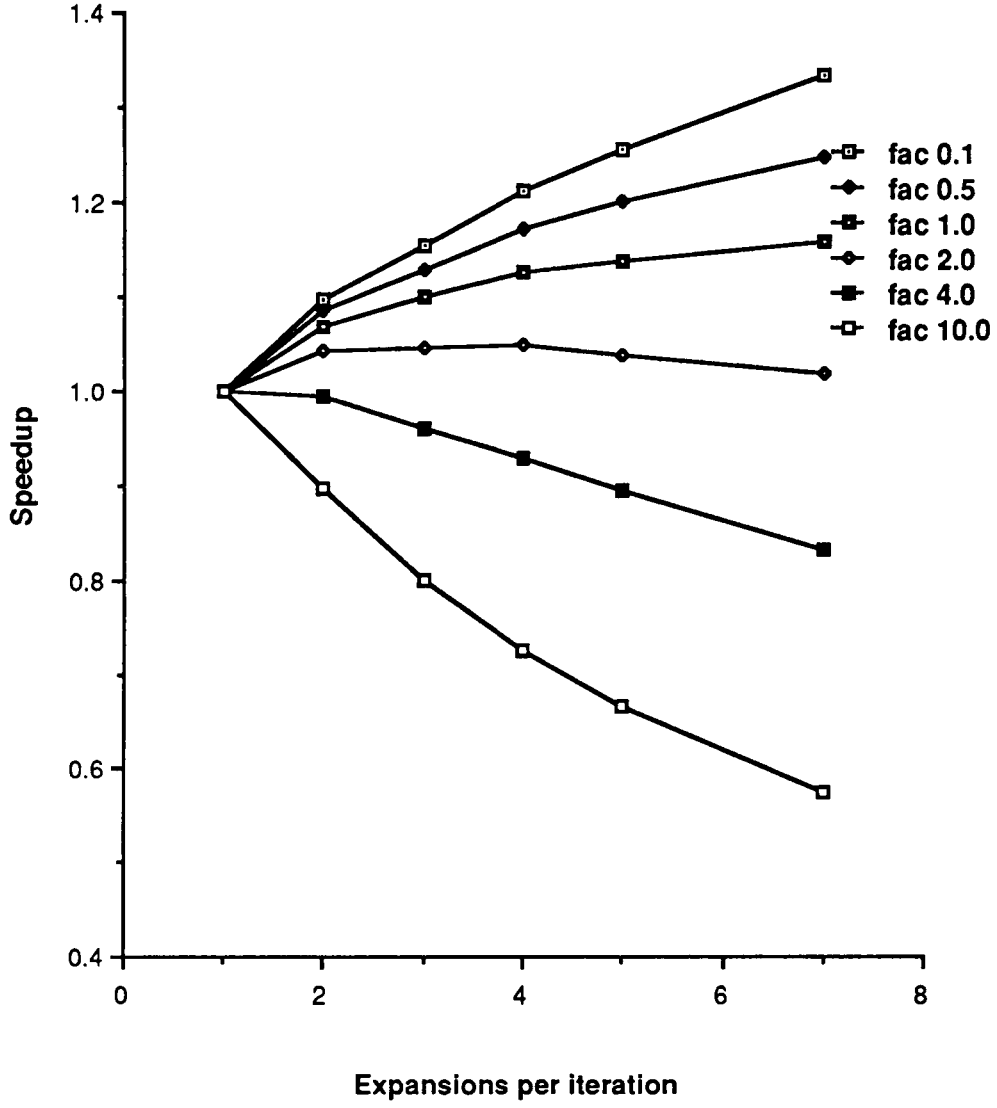

Figure 20. c_1-c_2 time speedup as a function of number of expansions per iteration at different factors of c_1. c_1 values at 1.0 are from table 4.2. Number of slaves: 7 Heuristic: Nearest neighbor.

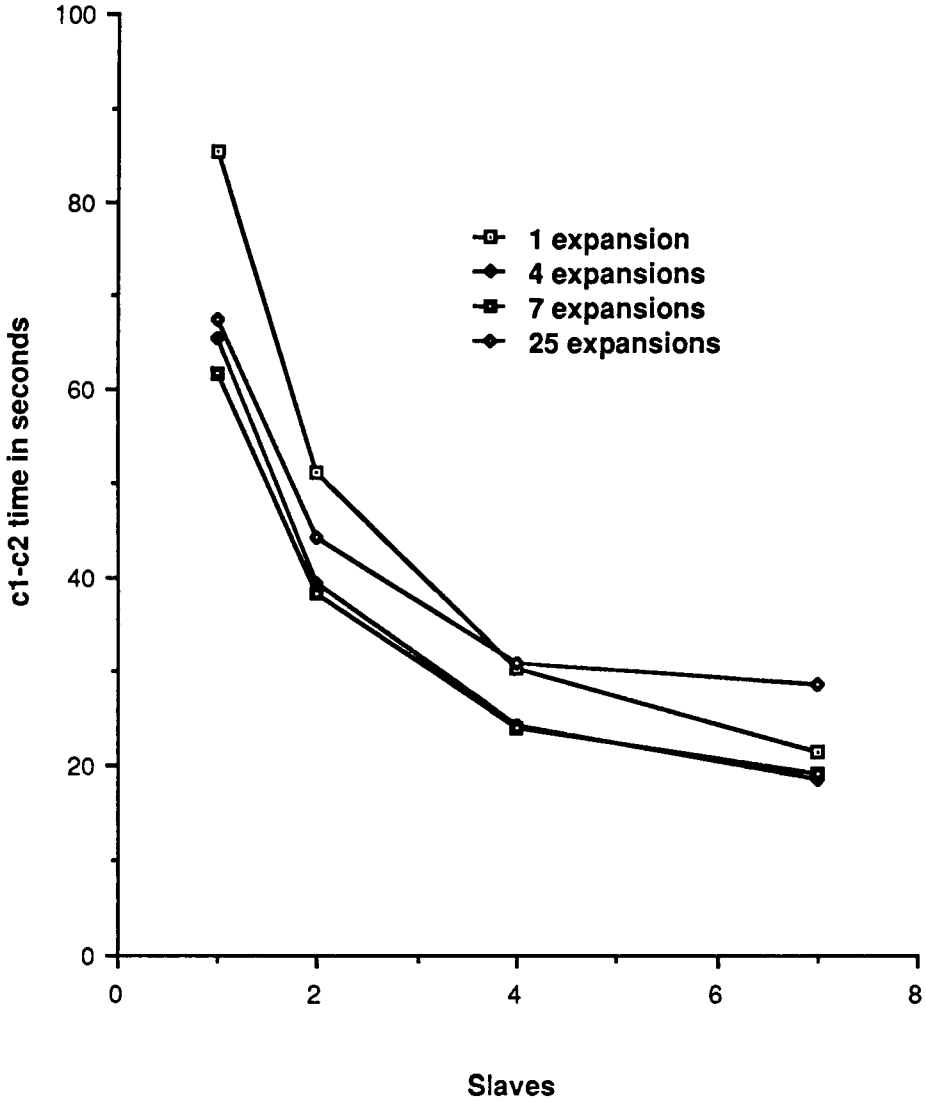

Figure 21. Average c_1-c_2 time as a function of number of slaves at different number of expansions per iteration. Heuristic: Minimum spanning tree.

considered are 1, 4, 7 and 25. Figure 21 also shows that if E is much higher than the optimal value, there may even be negative speedup relative to one expansion. For example, in case of 4 slaves, time taken at expansions 4 and 7 is much less than the time taken at expansions 1 and 25. It is also clear from the analysis of figure 21 that the decrease in time from 1 slave to 2 slaves is higher than the time decrease from 2 to 4 or 4 to 7 slaves. As the number of slaves increases, there may be a bottle neck in the master since the simultaneous computations increase. If there are more slaves, the master has to receive more data and sort it and put it in to the list. This becomes an additional overhead.

As pointed out earlier (it can also be noted from the figures 2, 4, and 21) the optimal point decreases with the increase in number of slaves. For example, the optimal point is 14 for 1 slave, 7 for 2 slaves, 5 for 4 slaves, and 3 for 7 slaves. As discussed earlier, this is because of the increased information generated by increase in the number of slaves. There is a need to increase the frequency of communication with the increase in the number of slaves. As a result, the optimal value decreases.

4.5 Variation of Solution Time for Individual Problems from Average Time

The time and speedup presented in the previous graphs are based on the average time taken over a large number of problems. Individual problems may

take more or less time than the average time. Figures 22 to 25 present such variations for different numbers of slaves. One hundred problems are solved using the minimum-spanning tree heuristic and the problem size is 10. The top and bottom curves represent the maximum and minimum amount of time taken by a problem in the set. The maximum time for each expansion tells that at least one of the problems in the set took that much time, and no problem in the set took more time than this maximum before the minimum cost solution is found. The minimum time tells that at least one of the problems in the set is solved in this short amount of time, and no problem in the set took less than this amount of time. These curves demonstrate the variations that could occur with different problems even if all other parameters including the number of cities are held constant. The center curve is the average time curve. Actually, the shape of both the maximum and minimum curves are pretty consistent with the average curve including at the optimal point. These graphs are presented to demonstrate that even though most of the time averages are close enough to represent a problem, there may be problem situations where this is completely out of proportions. The general behavior, however, remains the same.

4.6 Consistency of Smaller Problem Sets with a Large Problem Set

It is important to check the validity of average curves. In other words, are 100 problem instances sufficient to provide a good estimate of the true average time? Figure 26 presents the average of real times of two different problem sets

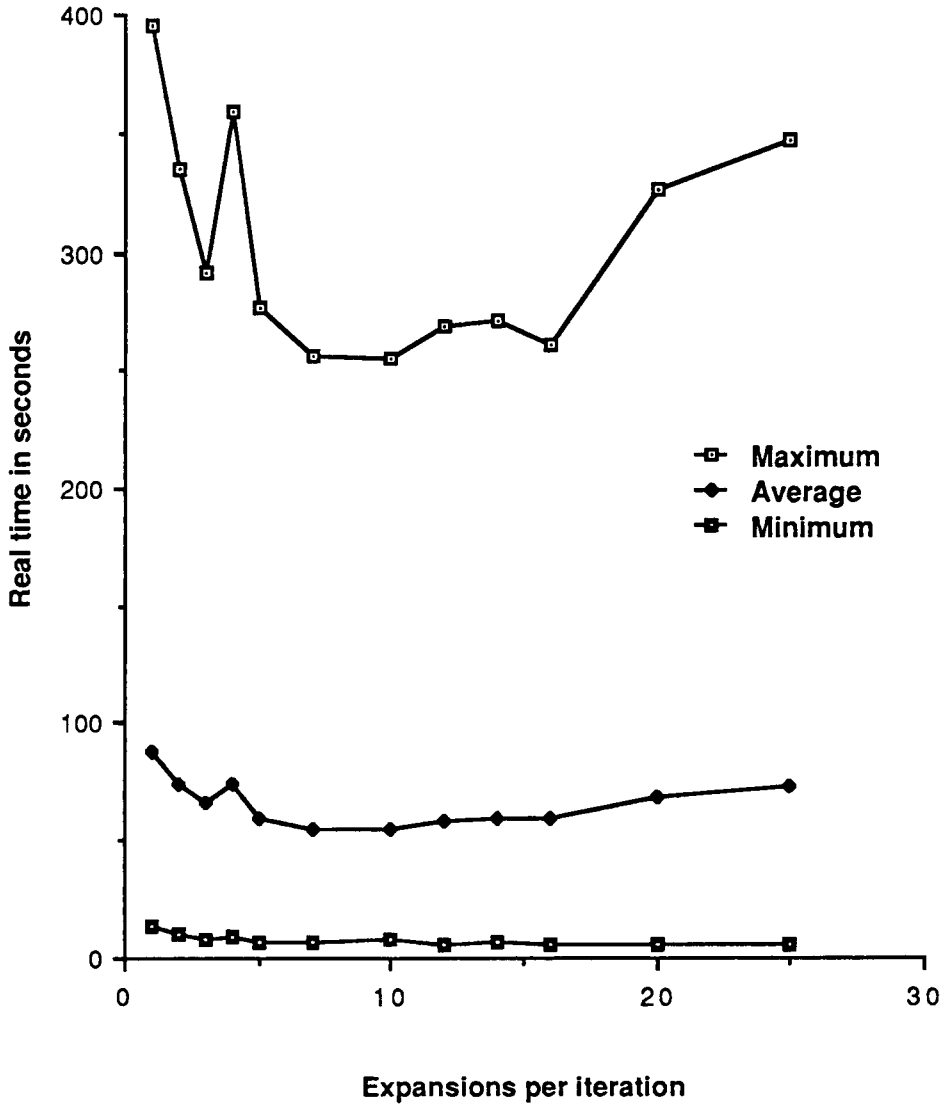

Figure 22. Maximum, Average, and Minimum real time as a function of number of expansions per iteration.
Number of slaves:1 Heuristic: Minimum spanning tree.

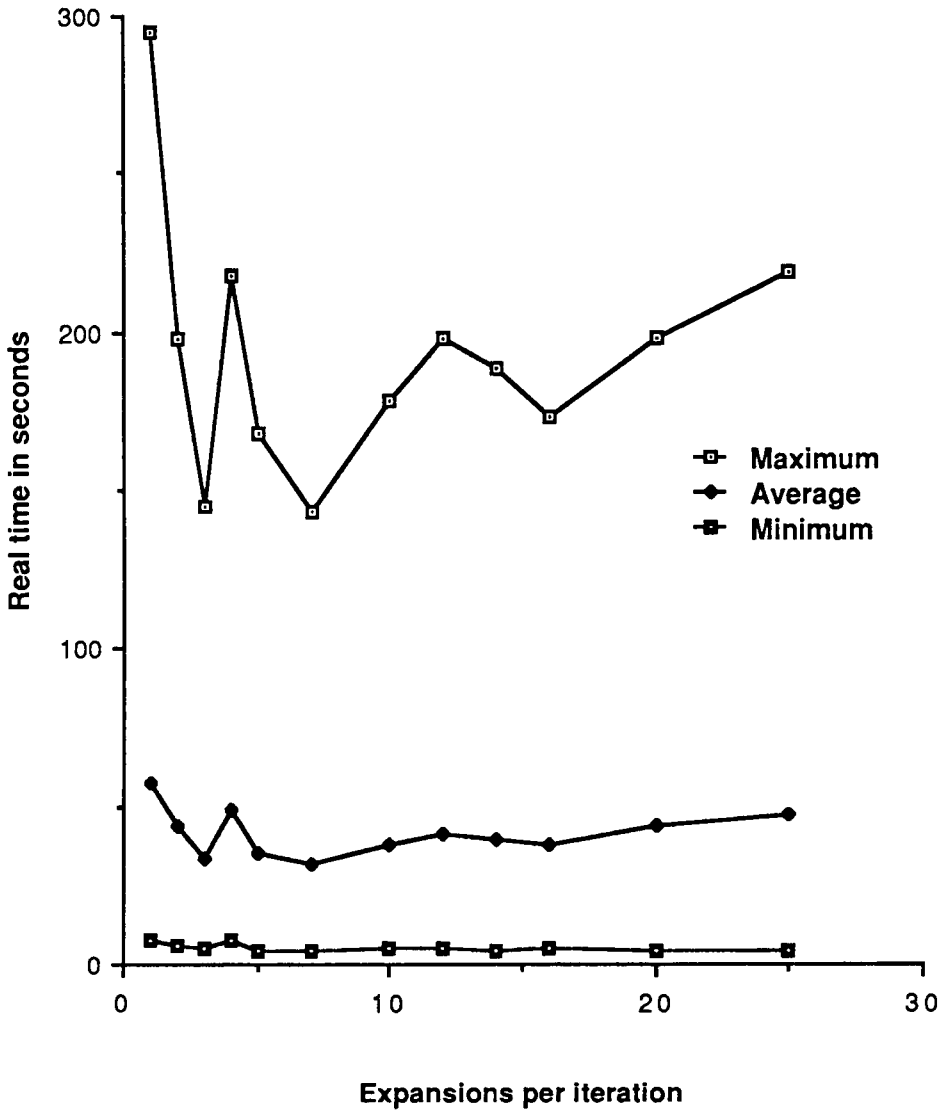

Figure 23. Maximum, Average, and Minimum real time as a function of number of expansions per iteration.
Number of slaves:2 Heuristic: Minimum spanning tree.

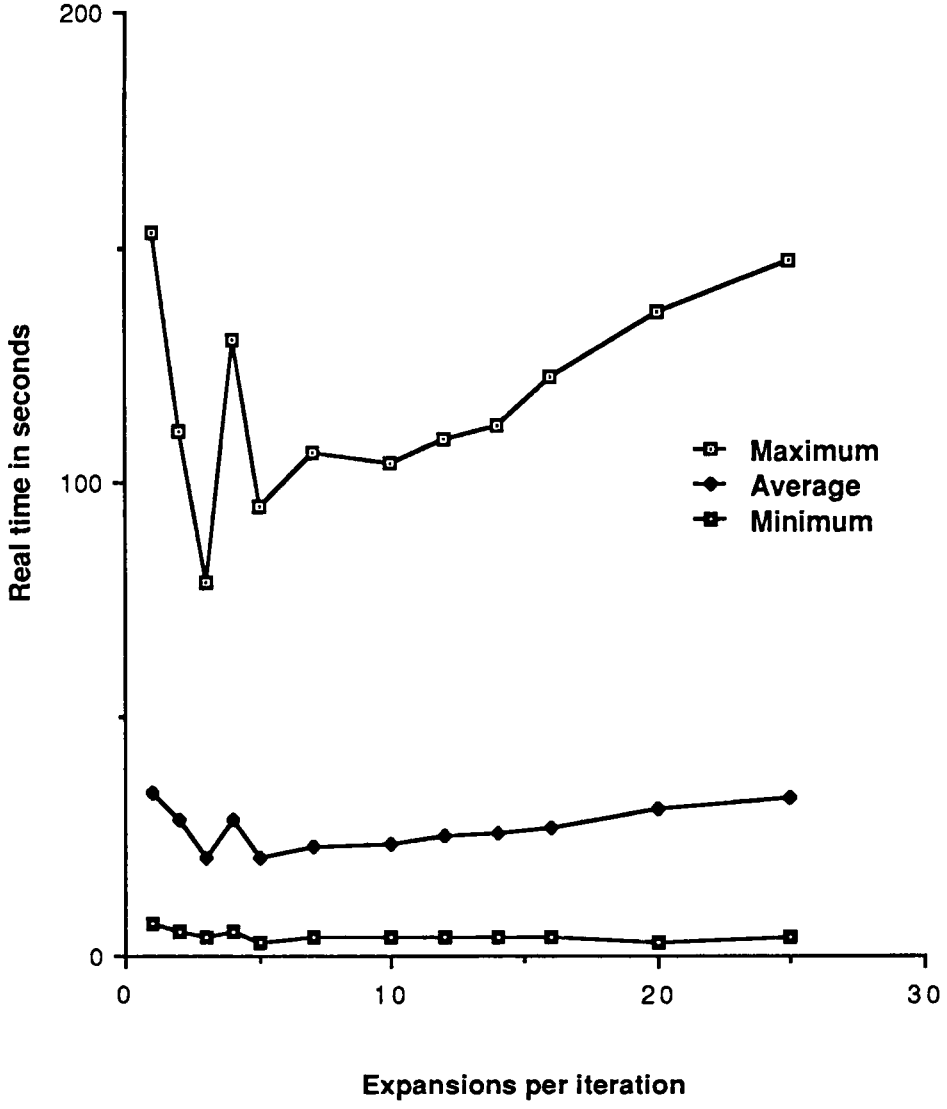

Figure 24. Maximum, Average, and Minimum real time as a function of number of expansions per iteration.
Number of slaves:4 Heuristic: Minimum spanning tree.

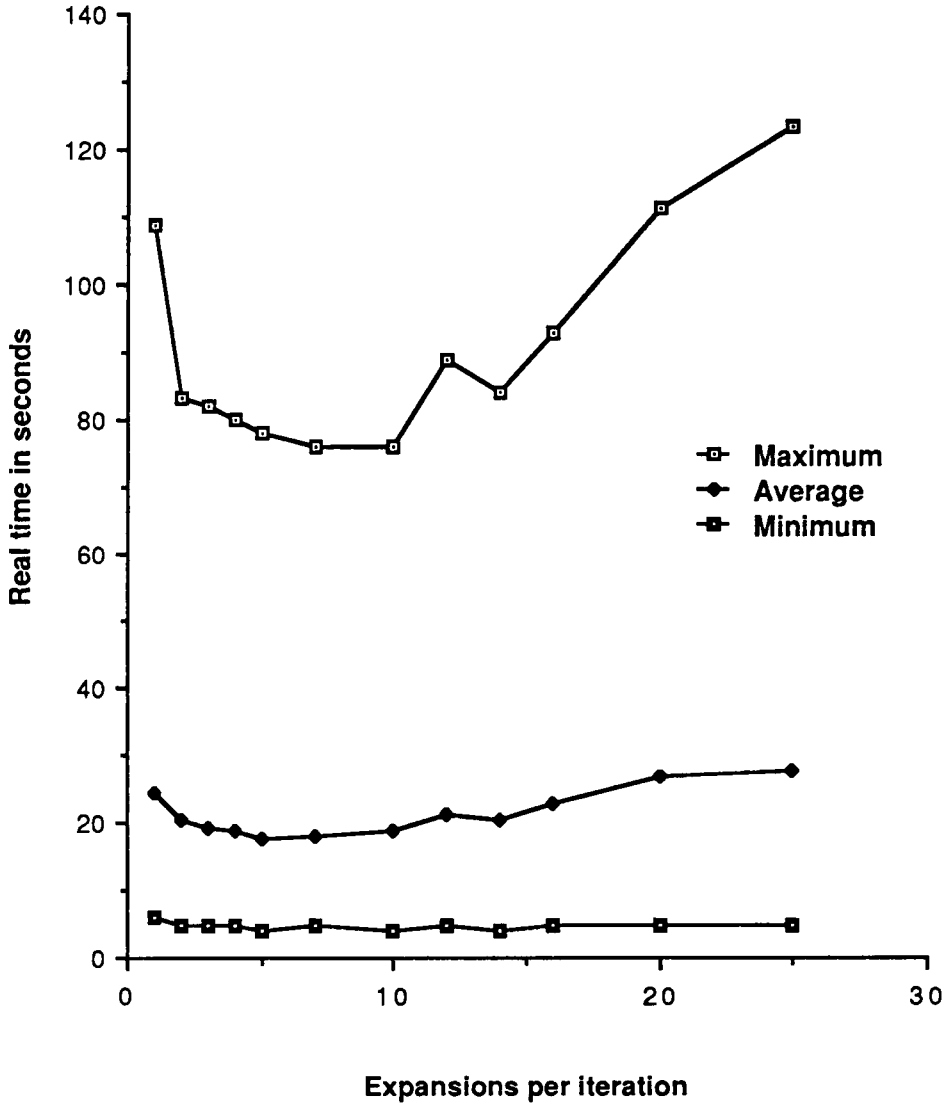

Figure 25. Maximum, Average, and Minimum real time as a function of number of expansions per iteration.
Number of slaves:7 Heuristic: Minimum spanning tree.

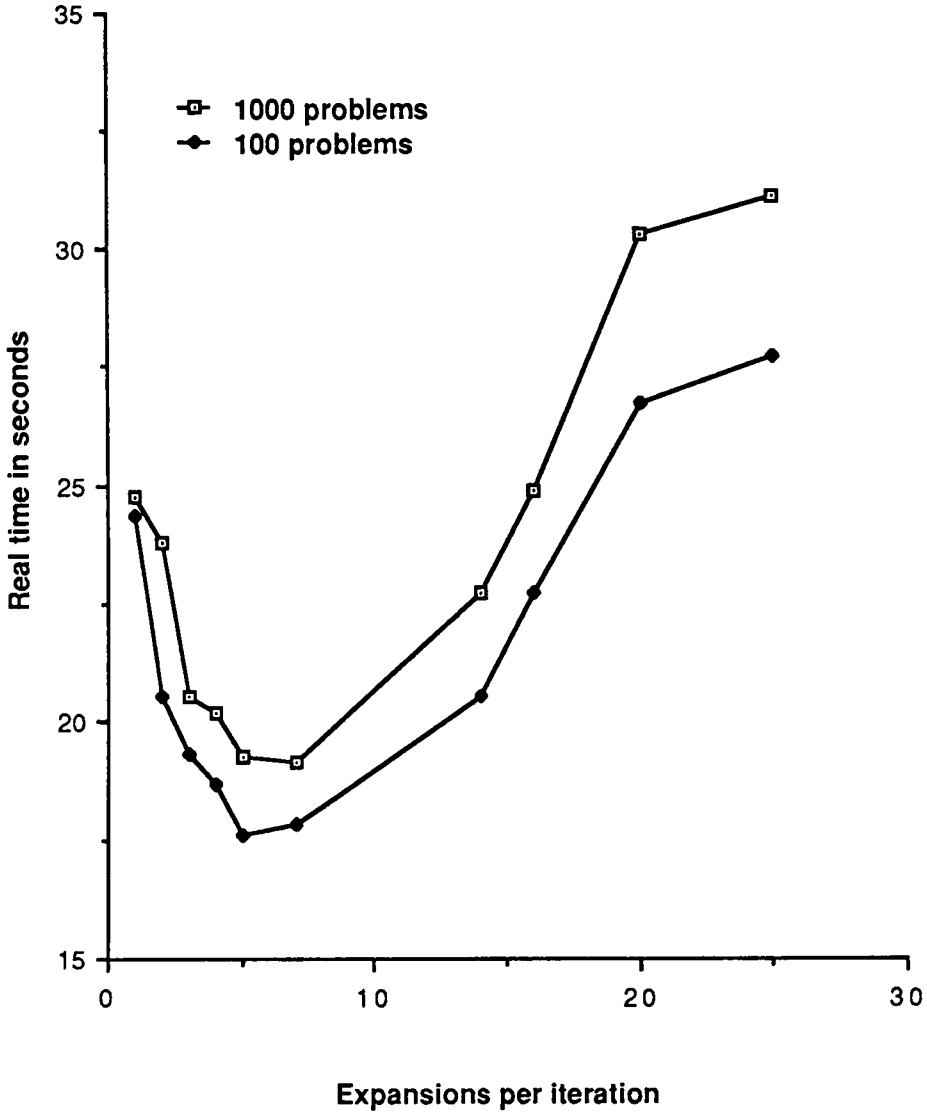

Figure 26. Average real time as a function of number of expansions per iteration for two different problem sets. Heuristic: Minimum spanning tree.

at 7 slaves. One set has 100 problem instances and the other one has 1000 problem instances. Averages are within 10% of each other. At some expansions, the difference is less than 1%. Perhaps, 10% difference is pretty reasonable especially when problem sets are completely different. Figure 27 presents the results similar to figure 26 but for c_1-c_2 time. The same results as in the case of figure 26 is observed. This improves our confidence in the results obtained earlier.

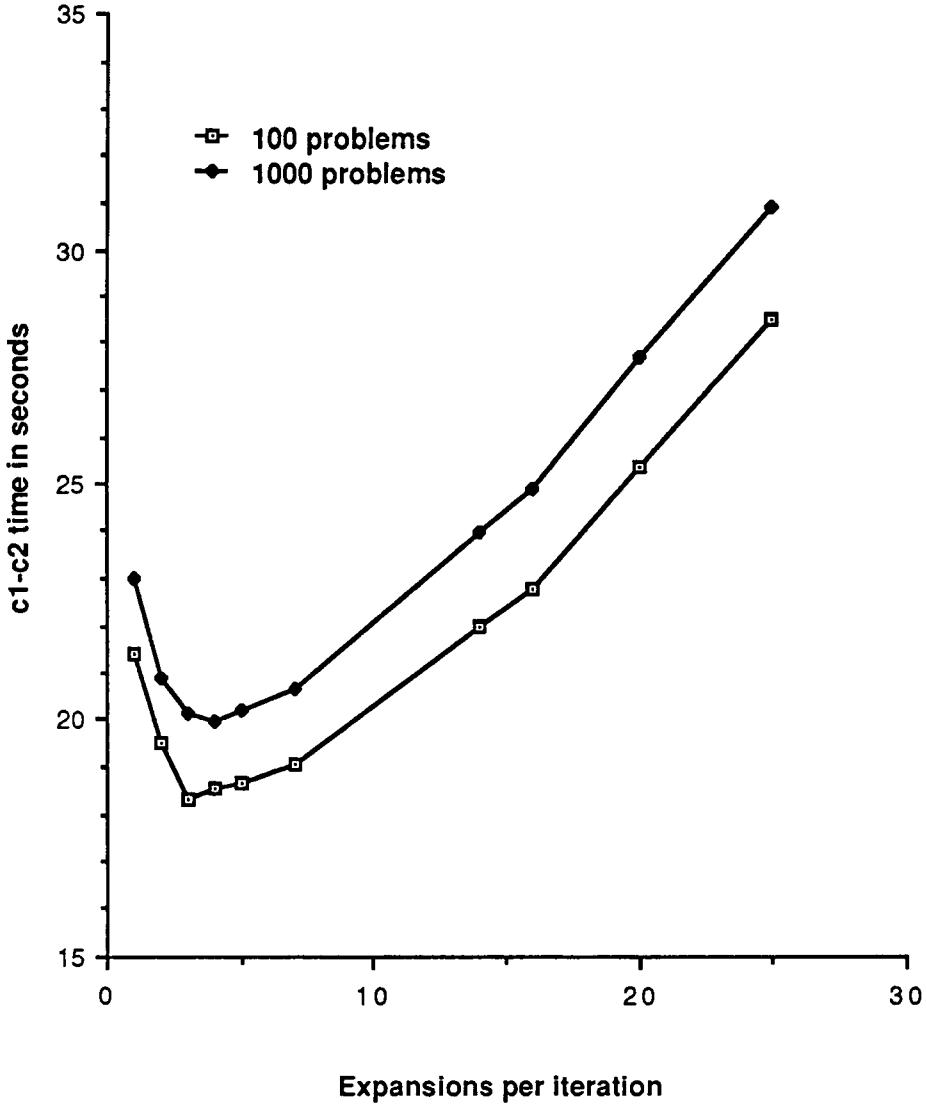

Figure 27. Average c_1-c_2 time as a function of number of expansions per iteration for two different problem sets. Heuristic: Minimum spanning tree.

Chapter 5

CONCLUSIONS AND FUTURE WORK

This chapter summarizes the achievements of this work and lists suggesions for future work. Both the conclusions and suggesions for future work are based on the results obtained as well as the experience gained in experimental design and execution. Evidence for such conclusions and suggestions are presented wherever appropriate.

5.1 Conclusions

The experiments showed that the number of expansions per iteration is a crucial factor in achieving higher speedups. The optimal number of expansions per iteration, at which the highest speedup is achieved, is a function of the time for each expansion (c_1) and communication time (c_2). The c_1 and c_2 in turn vary depending on the number of slaves. The experiments also clearly demonstrated that there should be a balance between total time spent expanding nodes and total time spent communicating. Note that the only way this balance can be achieved is by

controlling the number of expansions per iteration since only it is a software controlled parameter.

There may be some unexpected irregularities which might increase the elapsed time of a problem. This is evident from figure 1 especially when the number of expansions per iteration is equal to 4. With these minor exceptions, the behavior is quite as predicted. As we increase the number of expansions per iteration, the total elapsed time falls at first, then rises.

The empirical formulas developed for c_1 and c_2 are very useful for analyzing the large sets of data. It is clear from the results that as c_1 increases, the optimal value for the number of expansions per iteration decreases. This is because it becomes expensive to make more unnecessary expansions without updating the information with the other slaves and master. The effects of higher and lower c_1 are quite evident from figures 10 to 13. These show that the optimal value of E is inversely proportional to c_1.

It is also evident from the experiments that the optimal value E also depends on communication time, c_2. Unlike the case of c_1, the optimal value increases with increasing c_2. This is due to the fact that it is expensive to communicate frequently with higher c_2. Figures 5 to 8 show this. The optimal value is directly proportional to c_2. If the number expansions per iteration is much higher than the optimal value, there may even be negative speedups.

The number of slaves is another parameter that affects the optimal value. It is shown that as the number of slaves increases, the optimal value goes down. This is because as the number of slaves increases, there is more information generated by other slaves, thus providing a greater need to share information with the other slaves. This causes the optimal value to be lower with the increased number of slaves.

Figures 1 thru 4 show that speedups are not linear. As the number of slaves increases, the rate of increase in speedup decreases. However, the speedup is always positive at the optimal value for the cases presented in the results. Both c_1 and c_2 increase with the increasing number of slaves. The rise in these values is because as the number of slaves increase, there is more data to be transferred between the master and the slaves.

As expected, the average number of iterations never increases with the increasing number of expansions per iteration. It always decreases or stays constant. This is because of the synchronized nature of the algorithm. It is also evident that the decrease in number of iterations because of the increased number of expansions per iteration does not necessarily mean higher speedups. The decrease in number of iterations may not be significant and may not compensate for the cost of the increased number of expansions. This analysis for the number of iterations may not hold true if the algorithm is not synchronized.

It is quite clear from the comparison of the two different heuristics, minimum spanning tree and nearest neighbor, that the heuristic is one of the

crucial factors for speedup. A better heuristic makes pruning more efficient which in turn reduces the elapsed time causing better speedups. The same communication time between the master and the slaves (c_2) is observed for both heuristics but expansion time (c_1) in case of the minimum spanning tree heuristic is almost three times that of the nearest neighbor heuristic. However, the pruning efficiency of the minimum-spanning-tree heuristic makes the problems easier to solve than when the nearest-neighbor heuristic is used. As pointed out earlier, this is because of the nature of their computing technique in arriving to the heuristic value. For all cases, the minimum-spanning-tree heuristic is observed to be much superior to the nearest-neighbor heuristic.

Figures 22 to 25 show that there may be a large variation in computing time of individual problems from the average behavior. And also, the average time of smaller problem sets are compared with the larger problem sets to validate the general behavior of the smaller sets considered. The maximum difference is less than 10 percent. At some expansions, it is less as 1 percent. This is a good indication of the validity of the size of the problem set used.

Overall, the optimal number of expansions per iteration is found to be a crucial factor. The optimal value for the number of expansions varies depending on the number of slaves, expansion time (c_1) and communication time (c_2). This thesis focussed on the traveling salesman problem. The procedure for finding an optimal value does not give a general solution for other classes of problems. Some suggestions for how to do this are given in the next section for further work. The

main goal of this thesis is to show the importance of the optimal number of expansions and its dependency on various factors, and this is demonstrated clearly.

In summary, this thesis makes four contributions.

1. It shows through experiments that the choice for how many expansions to do per iteration can make a significant difference in the performance of parallel best-first search.

2. It suggests how certain parameters effect the optimal choice for how many expansions to do per iteration.

3. It provides a simple mathematical model to compute time per each expansion (c_1) and communication time (c_2). This mathematical model can be used as a rough predictor to obtain the optimal value for number of expansions per iteration (E). For this to be done, I(E) should either be extracted from data or needs to be estimated.

4. It provides a set of software for use in further experiments with parallel best-first search on a network of Suns.

5.2 Future Work

The accuracy of expansion time (c_1) and communication time (c_2) may be improved by incorporating the number of slaves (hosts) used in the formula. This may not be as easy as it seems. The key part here is separating the overhead

involved as the number of slaves increases. This could be achieved by running a large number of experiments and extracting new data. This improvement may even give a constant value for c_1 irrespective of number of slaves. The new relation may introduce a new parameter besides c_1 and c_2. The third parameter would take care of the increased overhead with the increased number of slaves.

New heuristics may be introduced for better performance. The two heuristics used in the experiments gave completely different time statistics. A few other heuristics can be tried and may even categorize the suitability of a particular heuristic to a particular class of problems. This may even lead to the development of an expert system to do the job.

This idea of developing an expert system can be further extended. Imagine a program that produces the optimal value for the number of expansions per iteration for a given class of problems. This goal can be achieved only with a very sophisticated expert system. Such an expert system should be capable of analyzing the different parameters such as number of slaves, communication and computation time and incorporating this information into the final decision. All the information needed to achieve this goal may not be readily available without conducting some experiments. However, once such an expert system is in hand, obtaining the optimal value can be automated even if it involves running some experiments. This achievement would be very useful in solving large class of problems.

The software developed for the work done in this thesis may also be improved. This may not be an easy task since the computation time and memory

requirements should be well balanced. A lot of effort was made to satisfy this requirement. Reducing memory requirements often causes the computation to rise drastically or vice versa. Effort was made to reduce the memory requirements drastically and a big rise in computational time was noticed. There are two lists maintained in the existing software to solve a problem. The first list is the open list which picks the first node in O(1) time and also has pointers to the second list. The second list contains the information about all the states. This may be improved by concatenating the two lists. It is possible to reduce the memory requirement if the nature of the problem is known in advance. If the values for *f*, *g*, and *h* are going to be smaller, then *short int* can be used instead of *int*. This drastically reduces the memory requirements. The other technique to get around the memory problem is by guessing a bound on the solution. This involves the following. First, an upper bound on the solution is guessed. The starting value for this upper bound should be as low as possible but reasonable. One suggestion for starting value is setting it to some constant (say 2) times the sum of the *N* (number of nodes) least arc costs. If the solution is not found within this bound, then the bound is incremented appropriately, and the process is repeated. This process can be repeated until either the solution is found or memory runs out.

Different classes of problems may also be tested. In the experiments done in this thesis, only traveling salesman problems were tested. The algorithm can be extended to different classes of problems with minor modifications to the software. Only the code for generating the children of a node, testing it at a goal, and computing the heuristic needs to be changed. The rest of the computation remains the same.

Confidence intervals can be defined to improve the reliability of the results. The confidence interval, a statistical technique, should be constructed in such a way that it gives certain confidence that the true value is within that interval. There are different techniques in the literature to achieve such an interval.

The model used in this thesis is synchronized. This means that a new set of nodes are sent to the slaves by the master only after it has received the newly generated information from all the slaves for its previous set. This algorithm may not be efficient for some applications. The algorithm can be asynchronized by sending the next best node to the slave immediately after the master has received the information from a slave. In other words, the master does not wait to send the data to a slave till it has received data from all other slaves. This may improve speedups.

The master-slave model used in this thesis uses point-to-point communication based upon stream sockets. Another model that DPUP supports is the broadcast model based upon the datagram sockets. The advantage of this is that there can be direct communication between the slaves as opposed to the master controlling all the slaves. This may avoid the bottleneck in the master if there are too many slaves. This can also overcome the disadvantage that certain operating systems limit the number of point-to-point connections. In this model, all processes are connected to one communication channel. Whenever a process sends a message, all the other processes can hear it. This communication may be used to update the goal value for pruning, or to terminate the processes, or to exchange the

best nodes between the slaves. However, the disadvantage of datagram is that delivery of message is not guaranteed.

Even though the best-first search guarantees the best solution if one exists, it is a very expensive process. If finding the best solution is not necessary, another algorithm may be used just to find the approximate values. These values are close to the best solution and often good enough for many problems. The effect of communication time on such approximation algorithms remains to be investigated.

BIBLIOGRAPHY

BIBLIOGRAPHY

1. Pearl, Judea, "Heuristics: Intelligent Search Strategies for computer problem solving., " Addison-Wesley publishing company, 1984.

2. Hart, Peter E., Nilsson, Nils J. and Raphael, Bertram, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths,", IEEE, Transactions of Systems and Cybernetics, Vol. SSC-4, No. 2, July 1968, pp. 100-107.

3. Nilsson, N. J., "Problem Solving Methods in Artificial Intelligence," McGraw-Hill, 1971.

4. Dechter, Rina and Pearl, Judea, "Generalized Best-First Search Strategies and the Optimality of A*," Journal of the ACM, Vol. 32, No. 3, July 1985, PP. 505-536.

5. Huyn, N., Dechter, R., and Pearl, J., "Probablistic Analysis of the Complexity of A*," Artificial Intelligence, 15, 1980, PP. 241-254.

6. Trienekens, H. W. J. M., "Parallel Branch and Bound on an MIMD system," Technical Report, CS-CU-354-87, Department of Computer Science, University of Colorado, Boulder, 1987.

7. Mohan, Joseph, "A Study in Parallel Computation--The Traveling Salesman Problem,", Technical Report, CMU-CS-82-136, Dept. of Computer Science, Carnegie-Mellon University, 1982.

8. Gardner, T., Gerard, I., Mowers, C., Nemeth, E., and Schnabel, R., "DPUP: A Distributed Processing Utilities Package," Technical Report, Dept. of Computer Science, Univ. of Colorado, Boulder, 1980.

9. Kindervater, G. A. P. and Trienekens, H. W. J. M., "Experiments with Parallel Algorithms for Combinatorial Problems," European Journal of Operations Research, 33, 1988, PP. 65-81.

10. Li, Guo-Jie and Wah, Benjamin W., "Computational Efficiency of Parallel Approximate Branch-and-Bound Algorithms,", IEEE Transactions on Computers, Vol. C-35, No. 6, June 1986, PP. 473-480.

11. Quinn, Michael J. and Deo, Narsingh, "An Upper Bound for the Speedup of Parallel Best-Bound Branch-and-Bound Algorithms," BIT 26, 1986, PP. 35-43.

12. Dunigan, T. H., "Performance of Three Hypercubes," Technical Report, ORNL/TM-10400, Engineering Physics and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN, 1986.

13. Dunigan, T. H., "A Message Passing Multiprocessor Simulator," Technical Report, ORNL/TM-9966, Engineering Physics and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN, 1986.

14. Vornberger, Oliver, "Fault Tolerant Parallelization of Branch-and-Bound Algorithms, " IEEE Computer Society, 1987, PP. 196-199.

15. Finkel, R. and Manber, U. "A Distributed Implementation of Backtracking," ACM Transactions on Programming Languages and Systems, Vol. 9, Apr 1987, No. 2, PP. 235-256.

16. Brassard, G. and Bratley, P., "Algorithms Theory & Practice," Prentice Hall, 1988.

17. Dunigan, T. H., "Hypercube Simulation on a Local Area Network," Technical Report, ORNL/TM-10685, Engineering Physics and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN, 1988.

18. Held, M. and Karp, R. M., "The Traveling-Salesman Problem and Minimum Spanning Trees," Operations Research 18 (1970), PP. 1138-1162.

19. Held, M. and Karp, R. M., "The Traveling-Salesman Problem and Minimum Spanning Trees: Part II," Mathematical Programming 1, 1971, PP. 6-25.

20. Fredman, M. and Tarjan, R. E., "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms," Journal of the ACM, vol. 34, no. 3, July 1987, PP. 596-615.

21. Vuillemin, J., "A Data Structure for Manipulating Priority Queues," Communication of ACM, 21, 4, Apr 1978. PP. 309-315.

22. Paige, Richard C. and Kruskal Clyde P., "Parallel Algorithms for Shortest Path Problems," IEEE, 1985, PP. 14-20.

23. Kumar, V. and Kamal, L., "Parallel Branch-and-Bound Formulations for And/Or Tree Search," IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. PAMI-6, No. 6, Nov 1984, PP. 768-778.

24. Kumar, V., "Integrating Knowledge in Problem Solving Search Procedures," Proceedings of ACM Annual Conference, 1984, PP. 5-10.

25. Kindervater, G. A. P. and Lenstra, J. K., "Parallel Computing in Combinatorial Optimization," Technical Report, OS-R8720, Center for Mathematical and Computer Science, Eramus University, Amsterdam, The Netherlands, 1987.

26. Kindervater, G. A. P. and Boxman, O. J., "A Queueing Network model for Analyzing a Class of Branch and Bound Algorithms on a Master-Slave Architecture," Technical Report, OS-R8717, Center for Mathematical and Computer Science, Eramus University, Amsterdam, The Netherlands,1987.

27. Li, Guo-Jie and Wah, B. W., "Coping with Anomalies in Parallel Branch-and-Bound Algorithms," IEEE Transactions on Computers, Vol. C-35, No. 6, June 1986, PP. 568-573

28. Lai, T. and Sprague, A., "A Note on Anomalies in Parallel Branch-and-Bound Algorithms with one-to-one Bounding Function," Information Processing letters 23, 1986, PP. 119-122.

29. Tanenbaum, A. S. and Renesse R. V., "Distributed Operating Systems, "Computing Surveys," Vol. 17, No. 4, Dec 1985, PP. 420-470.

30. Fox, B. L., Lenstra, J. K., Kan, Rinnooy, A. H. G., and Schrage, L. E., "Branching from the Largest Upper Bound," European Journal of Operations Research, 2, 1978, PP. 191-194.

31. Huang, S. and Davis, L. S., "A Tight Upper Bound for the Speedup of Parallel Best-First Branch-and-Bound Algorithms," Technical Report, CAR-TR-290, 1984, Center for Automation Research, University of Maryland, College Park, MD.

32. Lai, T. and Sahni, S. "Anomalies in Parallel Branch-and-Bound Algorithms," Communications of ACM, June 84, Volume 27, No. 6, PP. 594-602.

33. Karp, R. and Pearl, J., "Searching for an optimal Path in a Tree with Random Costs," Artificial Intelligence, 21, 1983, PP. 99-116.

APPENDIX

APPENDIX

$$F(c_1,c_2) = \sum_{i=1}^{N} [T_i - (I_i E_i * c_1 + I_i c_2)]^2 = 0$$

where

N = number of problems,

and T_i , I_i , and E_i are data collected from the i th experiment. Thus F is the sum of the squares of the differences between the measured time (T) and the predicted c_1-c_2 time.

To minimize this difference, the two equations,

$$\frac{\delta F(c_1,c_2)}{\delta c_1} = \sum_{i=1}^{N} \left(\frac{\delta}{\delta c_1} [T_i - (I_i E_i * c_1 + I_i c_2)]^2 \right) \quad -- \quad (1)$$

$$\frac{\delta F(c_1,c_2)}{\delta c_2} = \sum_{i=1}^{N} \left(\frac{\delta}{\delta c_2} [T_i - (I_i E_i * c_1 + I_i c_2)]^2 \right) \quad -- \quad (2)$$

Differentiating (1) and (2),

$$-2 \sum_{i=1}^{N} (T_i - I_i E_i c_1 - I_i c_2)(I_i E_i) \frac{\delta c_1}{\delta c_1} = 0 \quad \text{-- (3)}$$

$$-2 \sum_{i=1}^{N} (T_i - I_i E_i c_1 - I_i c_2)(I_i) \frac{\delta c_2}{\delta c_2} = 0 \quad \text{- -}$$

(4)

Equation (3) can be simplified as

$$\sum_{i=1}^{N} [T_i I_i E_i - c_1 (I_i E_i)^2 - (I_i^2 E_i) c_2] = 0$$

$$c_1 \sum (I_i E_i)^2 + c_2 \sum E_i I_i^2 = \sum T_i I_i E_i \quad \text{-- (5)}$$

Equation (4) can be simplified as

$$\sum_{i=1}^{N} [T_i I_i - c_1 E_i I_i^2 - I_i^2 c_2] = 0$$

$$c_1 \sum E_i I_i^2 + c_2 \sum I_i^2 = \sum T_i I_i \quad \text{-- (6)}$$

Equations (5) and (6) should be solved for c_1 and c_2.

To solve for c_1, multiply eq. (5) by $\sum I_i^2$ and eq. (6) by $\sum E_i I_i^2$.

Equation (5) becomes,

$$c_1 \sum (I_i E_i)^2 \sum I_i^2 + c_2 \sum E_i I_i^2 \sum I_i^2 = \sum T_i I_i E_i \sum I_i^2 \quad -- \quad (7)$$

Equation (6) becomes,

$$c_1 \sum E_i I_i^2 \sum E_i I_i^2 + c_2 \sum I_i^2 \sum E_i I_i^2 = \sum T_i I_i \sum E_i I_i^2 \quad -- \quad (8)$$

Solving equations (7) and (8) gives,

$$c_1 = \frac{\sum (T_i I_i E_i) \sum I_i^2 - \sum (T_i I_i) \sum I_i^2 E_i}{\sum (I_i E_i)^2 \sum I_i^2 - (\sum I_i^2 E_i)^2}$$

Equation (7) becomes,

$$c_1 \sum (I_i E_i)^2 \sum E_i I_i^2 + c_2 \sum E_i I_i^2 \sum E_i I_i^2 = \sum T_i I_i E_i \sum E_i I_i^2 \quad -- \quad (9)$$

Equation (8) becomes,

$$c_1 \sum E_i I_i^2 \sum (I_i E_i)^2 + c_2 \sum I_i^2 \sum (I_i E_i)^2 = \sum T_i I_i \sum (I_i E_i)^2 \quad -- \quad (10)$$

Solving equations (9) and (10) gives,

$$c_2 = \frac{\sum (T_i I_i E_i) \sum I_i^2 E_i - \sum (T_i E_i) \sum (I_i E_i)^2}{(\sum I_i^2 E_i)^2 - \sum (I_i E_i)^2 \sum I_i^2}$$

VITA

Purnachandra Rao Surapaneni was born on May 16, 1959, in Vijayawada, in the state of Andhra Pradesh, India. He spent most of his childhood in his native village Maredumaka of the Krishna district, in the state of Andhra Pradesh. He attended Andhra Loyola College, Vijayawada, and received his Intermediate diploma in 1975. He received a Bachelor degree in Mechanical Engineering in September, 1981 from the Siddaganga Institute of Technology, Tumkur, Bangalore University, India. In the fall of 1982, he came to the United States and received his Master's degree in Mechanical Engineering in March, 1985 from the University of Tennessee, Knoxville. He married Sailaja on may 10, 1985 in Vijayawada, India. He started the Master's program in Computer Science at the University of Tennessee, Knoxville in fall, 1986, and received his Master's degree in May, 1989. He is a coauthor of two papers entitled "Numerical Boundary Integral Analysis of Heat Transfer from Circular Tubes in a Semi-Infinite Medium" and "A Comparative Study of Project Estimating Tools."

The author is a member of the Association for Computing Machinery (ACM). He was inducted into the National Honor Society for the Computer Sciences, Upsilon Pi Epsilon (UPE), in 1988. He was the president of the Telugu Cultural Association in 1980-81 when he was a senior undergraduate student. The author is currently employed as a Senior Research Assistant at the Oak Ridge National Laboratories (ORNL), Oak Ridge, Tennessee.