Abstract

SIVAKUMAR, PABOLU. A Distributed Simulated Annealing Framework for Engineering Optimization (under the direction of Dr. John Baugh).

Engineering optimization problems are known to be difficult to solve using mathematical programming techniques because of large search spaces, complex objective and constraint functions, and, in many cases, their combinatorial nature. Simulated annealing is a well known heuristic optimization technique that has been used to solve a number of problems in discrete, non-differential, and combinatorial optimization and hence is suitable for solving such engineering optimization problems. However, computationally intensive problems are frequently encountered in the field of engineering optimization, in which case the use of simulated annealing can be prohibitively time consuming.

The objective of this thesis is to develop an object oriented framework that implements a distributed simulated annealing algorithm, which can be easily extended to solve computationally intensive engineering optimization problems. A distributed simulated annealing algorithm (**DSA Algorithm**) was developed and incorporated into a distributed simulated annealing framework called the **DSA Framework**. The framework defines interfaces, through which optimization problems can be modeled, utilizing a distributed computing framework, *Vitri*, to engage multiple desktop computers in a collective effort to solve problems.

The framework was used to solve a 40 variable knapsack problem as a benchmark problem to analyze the performance of the algorithm. The framework was also used to optimize support locations in a piping system subject to seismic loads. The DSA framework proves to be an efficient, fairly scalable tool that shows consistent reduction in execution time with increasing number of servers, thus proving to be a valuable tool in solving computationally

intensive engineering optimization problems.

A DISTRIBUTED SIMULATED ANNEALING FRAMEWORK FOR ENGINEERING OPTIMIZATION

by

Sivakumar Pabolu V

a thesis submitted to the graduate faculty of
north carolina state university
in partial fulfillment of the
requirements for the degree of
master of science

department of civil, construction, and environmental engineering

raleigh
August 2003

approved by:

Dr. John W. Baugh Jr.
co-chair of advisory committee

Dr. G. (Kumar) Mahinthakumar
co-chair of advisory committee

Dr. Abhinav Gupta

Biography

Sivakumar Pabolu V. was born on June 30th, 1979, in Tapeswaram, Andhra Pradesh. He was brought up in the town of Rajhmundry until 1987, when his family moved to Chennai, Tamilnadu. He pursued 8 years of education at Chennai, after which he moved with his family to Hyderabad, Andhra Pradesh in 1995. He received his Bachelor's degree in Civil Engineering from Indian Institute of Technology, Madras in 2001. He joined the Master's program in Civil Engineering at NCSU in fall of 2001 and has since been working as a Teaching Assistant with the department of Civil Engineering.

Acknowledgments

Firstly, I would like to thank my advisor Dr. John Baugh for his guidance and encouragement without which this research could not have been accomplished. I would like to express my gratitude to my co-chair Dr. Kumar Mahinthakumar for his help and guidance throughout my Masters education here at NCSU. I would also like to thank Dr. Abinav Gupta for his guidance and suggestions during the course of my Masters education. I would like to thank my colleagues Emily, Sunil and Parthee for their constant support and for being my family home away from home. I would also like to thank my roommates Vishal, Alok, Gaurav and Girish for all the fun times and friendship.

Last but not the least I express my deep gratitude to my parents for moulding me into the person I am today with their constant love and support.

Table of Contents

List of Tables **vi**

List of Figures **vii**

1 Introduction **1**
1.1 Organization of Thesis 3

2 Simulated Annealing **4**
2.1 Introduction 4
2.2 Number of transitions criteria 5
2.3 Temperature Reduction 6
2.4 Initial Temperature 6
2.5 Freezing Criterion 7

3 Distributed Simulated Annealing **8**
3.1 Single search path strategies 8
3.2 Multiple search path strategies 10
3.2.1 ASMCSPA 10

4 Implementation using *Vitri* **15**
4.1 Introduction 15
4.2 Distributed Computing Module in *Vitri* 15
4.3 Architecture of *Vitri* 17
4.3.1 Programming Environment 17
4.3.2 Class Structure 17
4.3.3 Salient Features 19
4.4 DSA implementation 20
4.4.1 Framework Details 21
4.5 Summary 26

5 Porting Vitri to a Parallel Platform **28**
5.1 *Vitri* 1.0 29

5.2 Porting 30
5.3 Test case 32

6 Results **38**
6.1 Knapsack problem 38
6.1.1 Solution Quality vs Time taken 40
6.1.2 Speedup 41
6.1.3 Efficiency 43
6.1.4 Number of Attempted and successful Transitions 45
6.2 Optimization of Support Location in a Piping System 46
6.2.1 Piping System Design 47

7 Conclusion **58**
7.1 Future Work 58

List of References **60**

A Using DSAFramework **63**
A.1 Extending the `Matter` Class 63
A.2 Input File 64
A.3 Executing the Program 66

List of Tables

6.1 Lumped Mass Distribution on Piping System 48
6.2 Support Capacities and Capital Costs 49
6.3 Average Capital Costs 50
6.4 Best Capital Costs 51
6.5 Time in Seconds for various cases 51
6.6 Speed up for various cases 52
6.7 Efficiency for various cases 57

List of Figures

2.1 Sequential simulated annealing algorithm 5

3.1 ASMCPSA Flow Chart 11
3.2 ASMCPSA algorithm 13

4.1 Pool of Tasks 16
4.2 Class Diagram of Distributed Computing Kernel 18
4.3 DSA framework Class Diagram 22

5.1 Vitri 1.0 Class Diagram 29
5.2 Solution Quality comparison of Distributed and Parallel runs 33
5.3 Solution Quality comparison of Distributed and Parallel runs 34
5.4 Solution Quality comparison of Distributed and Parallel runs 34
5.5 Speedup comparison of Distributed and Parallel runs 35
5.6 Speedup comparison of Distributed and Parallel runs 36
5.7 Speedup comparison of Distributed and Parallel runs 36
5.8 Efficiency comparison of Distributed and Parallel runs 37

6.1 Time vs Cost 41
6.2 Speedup for Knapsack problem 42
6.3 Speedup for Knapsack problem 43
6.4 Efficiency for Knapsack problem 44
6.5 Efficiency for Knapsack problem 44
6.6 Number of Attempted Transitions 45
6.7 Number of Successful Transitions 46
6.8 Pipe System Skeletal Model 48
6.9 Solution Quality vs Time 52
6.10 Solution Quality vs Time 53
6.11 Speedup 53
6.12 Model for information flow between Master - Slave processors for the 3 support case 55
6.13 Attempted Transitions 56
6.14 Successful Transitions 56

6.15 Efficiency 57

Chapter 1

Introduction

Engineering optimization problems, in particular those pertaining to design optimization, are very difficult to solve. They often involve large numbers of decision variables and even larger search spaces. The constraints and objective functions may be highly non-linear as they are often obtained by executing complex analysis models. In addition, when the decision variables are discrete, as when selecting structural members from standard shapes, these problems come under the challenging class of combinatorial problems. Due to these concerns, engineering optimization problems are often not amenable to mathematical programming techniques, thereby dictating the use of heuristic optimization techniques.

Modern heuristic optimization techniques such as simulated annealing, genetic algorithms and tabu search have been shown to work very well for a wide variety of problems. While each has problem domains to which it is particularly suited, simulated annealing has been shown to work well on a variety of problems of interest to civil engineers, including vehicle routing and scheduling [10], structural truss design optimization [9], water distribution network design optimization [6] and structural optimization [16].

Distinctive features of simulated annealing include its hill climbing ability and its ability to transition from a global into local search strategy as the computation unfolds. It is well suited for problems with a well defined neighborhood structure and is particularly useful in engineering optimization problems since it proceeds incrementally from one solution to another by making a single change at a time. Even though the simulated annealing algorithm does not guarantee an optimal solution, it often produces good solutions in considerably less time than mathematical programming techniques.

Most engineering optimization problems are computationally intensive. During the course of its execution, the simulated annealing algorithm evaluates a number of candidate solutions, each of which may involve executing complex analysis models. For such problems, simulated annealing can prove to be prohibitively slow. However, advances in high performance computing have made even these problems practical in many situations.

Parallel and distributed computing are viable options to speed up the optimization process by efficiently distributing work among a number of cooperating processors. Parallel programming provides the benefit of powerful processors connected with high-speed, tightly coupled networks. Thus, parallel computing is viable for solving both coarse grained and fine grained optimization problems. However, not all programming languages are optimized to work well on parallel architectures, and parallel machines are generally few in number and of limited availability.

In contrast, distributed computing provides the benefit of utilizing easily accessible desktop computers connected by an Ethernet connection to perform computationally intensive tasks. However, the network connections between processors are much slower than those of a parallel machine. Thus, distributed computing is more suitable for solving problems of the medium- to coarse-grained variety. Thus a parallel implementation of the simulated annealing algorithm, which can be executed on a cluster of desktop computers, can be a powerful tool in solving a number of engineering optimization problems.

The objective of this thesis is to develop an object oriented framework that implements a distributed simulated annealing algorithm, which can be easily extended to solve computationally intensive engineering optimization problems. To achieve this, a Distributed Simulated Annealing algorithm (**DSA algorithm**) was developed based on the work by Lee *et al.* [15]. A Distributed Simulated Annealing framework (**DSA framework**) has been implemented which incorporates the DSA algorithm and defines interfaces through which optimization problems can be modeled. The distributed computing component of the framework utilizes a distributed computing framework, *Vitri* [11], to engage multiple desktop computers in a collective effort to solve problems.

Apart from developing the framework to work on a cluster of machines, *Vitri* was ported to a parallel machine. The TCP/IP sockets based communication used in *Vitri* was replaced with **mpiJava** [1] based MPI calls in the parallel version. Thus, a program developed to use the DSA framework can be used on a parallel machine with few modifications.

1.1 Organization of Thesis

The properties and the formulation of a sequential simulated annealing algorithm are presented in Chapter 2. Introducing parallelism into the simulated annealing algorithm and the working details of the DSA algorithm are discussed in Chapter 3. A description of *Vitri*, which provides the underlying distributed computing capabilities, are described in Chapter 4, along with implementation details of the DSA framework. Porting *Vitri* to a parallel platform is discussed in Chapter 5. Finally, the performance of the DSA Framework when applied to a knapsack problem and a structural optimization problem is discussed in Chapter 6.

Chapter 2

Simulated Annealing

2.1 Introduction

Simulated annealing is one of the popular heuristic optimization techniques. Since its introduction by Kirkpatrick *et al.* [12], it has been used in solving various classes of optimization problems including discrete, non-differentiable and combinatorial optimization problems. Distinctive features of the Simulated Annealing (**SA**) algorithm include its hill climbing ability and its ability to transition from a global into local search strategy as the computation unfolds.

In order to use SA, the solution to the problem is typically represented as an array, list, or other well-defined sequence of variables, which is called a **Solution representation**. The algorithm starts with an initial solution and attempts are made to make a random modification, called a **Solution Transition**, to this solution representation based on some rules. The rules used to make a transition are problem dependent and are defined to cover the search space effectively. A solution transitions which leads to an improvement in objective value is called an improving transitions and one which leads to deterioration in objective value is called a non-improving transition. The simulated annealing algorithm also uses a quantity called **Temperature**, which is used to determine the probability of acceptance of non-improving temperature transitions. Temperature is a non-dimensionl quantity, used to drive the algorithm gradually towards finding improving solutions.

As described in Figure 2.1, the algorithm starts by assuming a reasonably high temperature. A number of solution transitions are attempted at each temperature, with those improving the objective function being immediately accepted. If the transition is objective non-improving, it is accepted with a probability which is a function of the current temperature and the change in objective function value. As shown in step 6 of Figure 2.1, after completion of transition attempts at a given temperature, the temperature is reduced by a constant factor. This process continues until the SA algorithm is frozen.

Step 1 start with a randomly generated solution s
Step 2 select an initial temperature T
Step 3 perform solution transition from s to s'
Step 4 accept s' with a probability of $min\left[1, exp\left(\frac{-\Delta}{T}\right)\right]$, where $\Delta = cost(s') - cost(s)$
Step 5 repeat step 3 until number of transitions criterion is satisfied
Step 6 $T = \alpha T$
Step 7 repeat steps steps 3–5 until freezing criterion is achieved

Figure 2.1: Sequential simulated annealing algorithm

2.2 Number of transitions criteria

At each temperature, the number of attempted transitions (n_a) and number of successful transitions (n_s) are monitored. The number of transitions criteria are said to be satisfied if one of the following is true

- $n_a >$ Maximum number of allowable transitions at each temperature ($N_{all\ trans}$)
- $n_s >$ Maximum number of successful transitions allowable at each temperature ($N_{succ\ trans}$)

The annealing scheme is said to have achieved a quasi equilibrium state if one of these two criteria is satisfied. Generally, $N_{all\ trans} \approx$ neighborhood size of the solution representation, and $N_{succ\ trans} = \frac{1}{2} N_{all\ trans}$. At each temperature, solution transitions are attempted until quasi equilibrium state is achieved. This process of attempting transitions at a particular temperature until the transitions criteria are satisfied is called a **temperature-run**.

2.3 Temperature Reduction

Upon the completion of a temperature-run, the subsequent temperature-run is performed at temperature T' less than T, where $T' = \alpha T$. The temperature reduction factor, α, varies from problem to problem and has a value between 0.85 to 0.97.

2.4 Initial Temperature

In SA, the search process is initially performed in a random manner accepting all the attempted solution transitions. However, as the algorithm proceeds, the probability of acceptance of non-improving solution transitions is progressively reduced. This initial acceptance of all attempted transitions allows substantial portions of the search space to be explored and gives a global scope to the search process.

At a given temperature T, the probability of acceptance of a solution transition is

$$\chi = min\left[1, exp\left(\frac{-\Delta}{T}\right)\right]$$

where Δ is the change in objective value due the solution transition. A low initial temperature implies less probability of acceptance for non-improving solution transitions. Therefore, it is preferable to use an initial temperature that is high enough to permit an acceptance rate of about 90% initially.

This initial temperature can be determined either by trial and error or by using methods which sample behavior of the solution to perturbations to arrive at the initial temperature. Baugh *et al.* [10] describe a method to determine the initial temperature. This method has been used for initial temperature selection for the various problems to which the DSA algorithm has been used.

In order to determine the initial temperature, a number of random transitions (N_a) are performed on the solution. Of these N_a attempts, let m_1 be the number of objective improving transitions and m_2 be non-improving transitions. Let ΔC^+ be the average increase in cost over m_2 non-improving transitions. The acceptance rate at a given temperature T can be represented by

$$\chi = \frac{m_1 + m_2 exp\left(-\frac{\Delta C^+}{T}\right)}{m_1 + m_2} \tag{2.1}$$

This formula can be re-written as

$$T = \frac{\Delta C^+}{ln\left[\frac{m_2}{m_2\chi - m_1(1-\chi)}\right]} \tag{2.2}$$

The values of $m_1, m_2 and \Delta C^+$ found after N_a attempted transitions are substituted in Equation 2.2, with an acceptance rate of 90% chosen for χ. However, it is important to have a suitably large number of attempts to arrive at the initial temperature. Test runs of this initial temperature selection method were performed to find the initial temperature for a 40 variable knapsack problem. Observations from the results of these runs indicate that a value of about 5 times the neighborhood size of the solution representation appeared to be a reasonable value for N_a.

2.5 Freezing Criterion

Figure 2.1 shows that, as the algorithm proceeds, the reducing temperature diminishes the probability of acceptance of non-improving transitions. Thus at low temperatures a substantial number of attempted transitions are unsuccessful. If no successful transitions are performed in N_{frozen} number of subsequent temperature-runs, the annealing scheme is deemed frozen and terminated.

Although SA algorithms are known to work well for various types of optimization problems, their execution time can be considerably high, particularly when function evaluation is costly. In such situations, parallelizing the SA algorithm can be of considerable benefit. The sequential SA algorithm, presented above, forms the basis for the Distributed Simulated Annealing algorithm used in implementing the DSA framework. The subsequent chapter discusses the various methods for parallelizing this algorithm and the issues arising in the process.

Chapter 3

Distributed Simulated Annealing

The simulated annealing process consists of a sequence of transitions from one solution (state) to another based on a probability. Thus the search path followed by a Sequential Simulated Annealing **(SSA)** algorithm may be viewed as a Markov chain [15]. There have been a number of attempts to use the time saving benefits of parallel computing by parallelizing this search process. Several approaches have been taken by researchers to utilize parallelism at various levels of the simulated annealing process [2, 16, 19, 20, 21, 22]. All parallelizing strategies are based on the idea that multiple processors can be used to make concurrent moves in decision space at various degrees of independence from each other. These strategies can be broadly classified into either single search path (single Markov chain) or multiple search path (multiple Markov chain) strategies. Lee *et al.* [15] have performed extensive tests in analyzing the performance of various approaches for parallelizing simulated annealing including single and multiple Markov chain models. The Distributed Simulated Annealing algorithm (**DSA algorithm**) used to implement this framework has been based on the Asynchronous Multiple Markov Chain Parallel Simulated Annealing **(ASMCPSA)** model proposed by Lee *et al.* In subsequent sections, the various approaches for parallelizing simulated annealing and the rationale for selecting ASMCPSA are explained.

3.1 Single search path strategies

In a single search path strategy, a master processor initializes a solution, and disburses it to a number of slave processors. The slave processors each attempt a transition on this solution and report the new solution and change in cost to the master processor. The master processor decides which of these candidate solution transitions to accept and broadcasts the new solution to all the slave processors. Single search path (Markov Chains) are based on the principle that the current solution (incumbent) of all the processors is the same. Therefore, in this strategy more options are explored for a solution transition at each step. This process is not efficient because

- It involves a substantial communication overhead as communication is needed at every single transition attempted by the slave processor.
- It involves a substantial synchronization overhead, as the master processor waits on all the slave processors to report their respective solutions before broadcasting the next solutions. This constitutes a sizable portion of idle time for the slave processors.
- This approach tightly couples all the participating slave processors, where none of the processors are allowed to explore search space independent of other processors for more than one solution transition. Further parallelism can be exploited by allowing each processor more freedom to perform solution transitions and enforcing as little communication and synchronization as possible.

As an improvement upon this process, a certain amount of relaxation can be given on the synchronization, such as synchronizing the candidate transitions after a number of attempted transitions by the slave processor, where the slave processor can choose to accept or decline the transitions using the Boltzmann probability.

Another approach, suggested by Nabhan *et al.* [19], is to follow the simple single search path strategy. However, at low temperatures, the communication and synchronization requirements on the slave processor are reduced, where the slave processors are allowed to attempt transitions, decide whether or not accept transitions based on the Boltzmann probability, and required to report their solution to the master processor only if a solution transition is accepted by the slave processor. Another approach suggested is to allow for low communication toward the end of the annealing scheme, when slave processors communicate with the master processor only when a move is accepted rather after a fixed number of attempts. Many such approaches have been proposed that introduce relaxation on communication and synchronization at various levels. However, these still follow the notion of a single search path and leave scope for further exploitation of parallelism.

3.2 Multiple search path strategies

Though there are methods to reduce communication and synchronization overheads associated with single Markov chain models, they involve communication and synchronization a number of times in each temperature-run. Given the fact that most SAs involve a large number of temperature-runs, such frequent communication still forms a sizable part of the parallel algorithm.

It would be desirable to allow each processor to perform search unhindered for a number of temperature-runs and synchronizing with the master processor at much less frequent intervals than is done in single search path strategies. Therefore a good alternative would be to allow each processor to execute Markov chains, each of which performs a search process similar to SSA independently for a number of temperature runs after which their solution is reported to the master processor. Synchronization may or may not be enforced on this reporting procedure.

Lee *et al.* have observed that such loosely synchronized multiple Markov chains show an order of magnitude improvement over any variant of single Markov chain in terms of solution quality, communication and synchronization overheads. They have also observed that among various types of multiple search path algorithms, the Asynchronous Multiple Markov Chain Parallel Simulated Annealing algorithm shows the best performance in terms of solution quality, communication and synchronization overheads. One such strategy suggested by them, ASMCPSA, has been used in this thesis to develop the DSA algorithm.

3.2.1 ASMCSPA

Lee *et al.* propose an Asynchronous multiple Markov Chain Parallel Simulated Annealing algorithm, in which a number of search paths (Markov Chains) are explored concurrently by participating slave processors. Figures 3.1, 3.2 presents a details of this algorithm as it is executed on the master and slave processors. Each slave processor executes an SSA-like search process (Markov Chain or MC) where each processor performs the tasks of making solution transitions, accepting candidate solutions, and reducing the temperature after a number of attempted transitions.

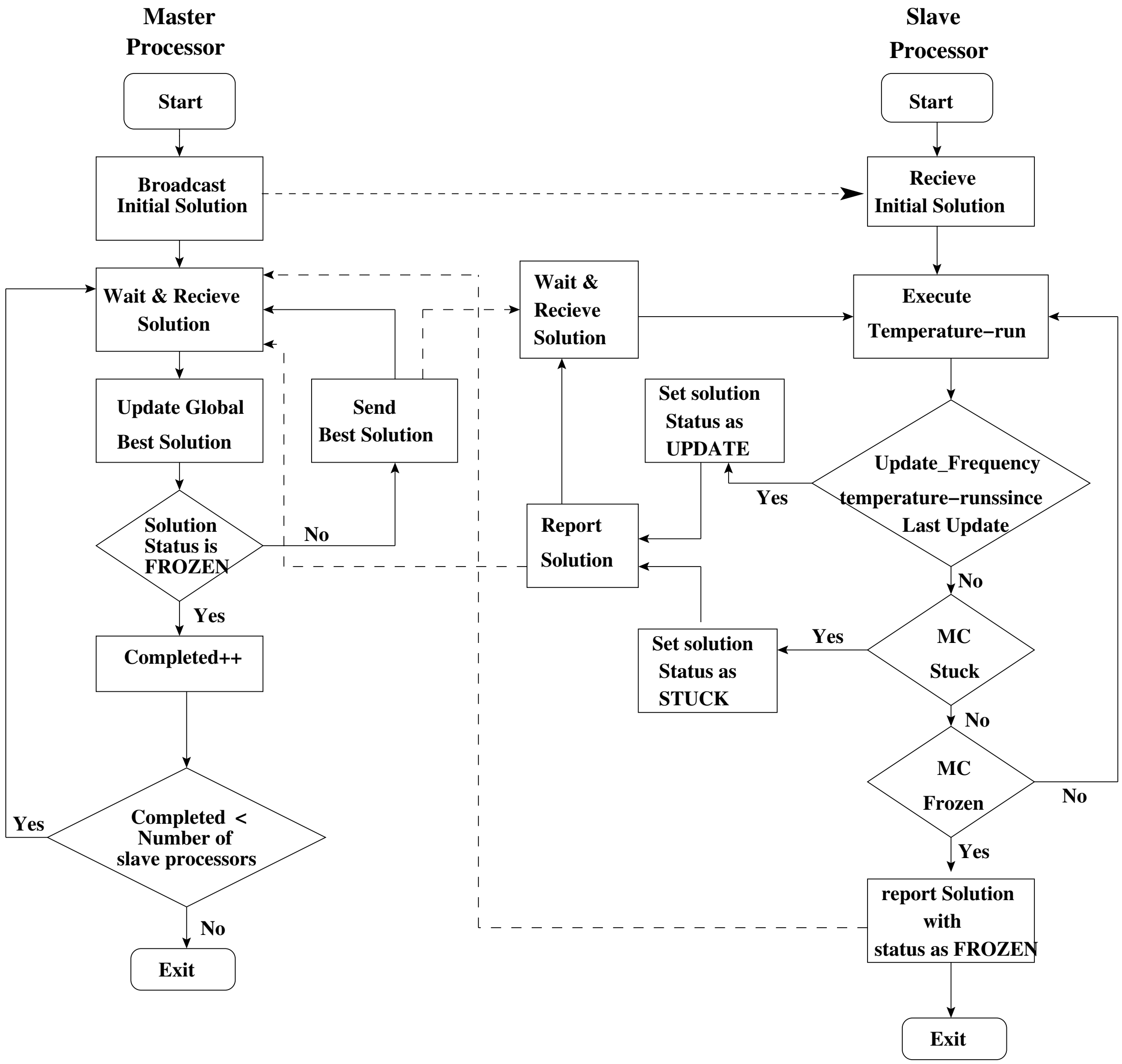

Figure 3.1: ASMCPSA Flow Chart

The annealing scheme parameters like initial temperature and temperature reduction factor used for these MCs are the same as those used for an SSA. However, the maximum number of attempted transitions ($N'_{all\ trans}$) at each temperature in each MC is a fraction (β) of those for an SSA. This reduction in number of attempted transitions at each temperature is a way to approximately distribute the work performed by an SSA at each temperature among a number of slave processors. Thus, with n slave processors,

$$N'_{all\ trans} = \beta N_{all\ trans}, where \tag{3.1}$$

$$\beta = \frac{1}{n} \tag{3.2}$$

The parameter β is the **Step-down factor**. Lee *et al.* propose other functions to arrive at values for β such as $\frac{1}{\sqrt{n}}, \frac{1}{log_2 n+1}$. They also mention that using these functions can lead to a trade off either in terms of solution quality or speedup. For this thesis, the value of $\frac{1}{n}$ has been used as the step-down factor.

Apart from independent execution of MCs, each slave processor reports the best solution found to the master processor after every $N_{update\ freq}$ number of temperature-runs (refer to Section 2.2). The master processor maintains the best solution reported by all the processors at all times and upon receipt of a solution from a slave processor overrides the current best with the new solution if it is better, otherwise it returns the best solution to the slave processor. This process is asynchronous and the master processor handles the reported solutions as and when they are received. This process by which the master processor is informed about the best solution found by the slave processors is called an **Update** process. After receiving an update, the master processor sends a copy of the current best solution to the slave processor, and the slave processor continues the execution of the MC from where it was halted.

As discussed in Section 2.5, an SSA is deemed frozen if no successful transitions are performed in N_{frozen} temperature-runs. Similarly, an MC is deemed frozen if no successful transitions are performed in N_{frozen} temperature-runs.

Apart from being frozen, ASMCPSA also has the notion that an MC is **Stuck**, where the MC is unable to make any transitions in N_{stuck} number of temperature-runs.

Each MC performs a fraction of the total number of attempted transitions as compared to the SSA. This implies that at low temperatures, there is a higher likelihood of an MC not being able to make even one successful transition as compared to SSA. There may be cases where at a certain temperature an MC is deemed frozen, while a corresponding SSA may not be frozen at such temperature.

Therefore the notion of being stuck is introduced, where, if the MC is unable to make any successful transitions in N_{stuck} number of temperatures, it is allowed to request an

Master Processor

Step 1 Initialize and broadcast initial solution to all the slave processors

Step 2 Listen for reported solutions.

Step 3 If received solution better than global best solution, replace global best with received solution.

Step 4 If received solution status is UPDATE or STUCK, return global best solution to slave processor.

Step 5 If received solution status is FROZEN, increment $N_{complete}$.

Step 6 Repeat Steps 2 to 5 while $N_{complete} <$ number of slave processors

Slave Processor

Step 1 Receive initial solution s from Master processor

Step 2 Start an MC with initial Temperature T_o

Step 3 Execute a Temperature run (Steps 3 to 6 of Figure 2.1)

Step 4 If MC Frozen, report solution to Master processor with status FROZEN and exit.

Step 5 If MC Stuck, report solution to Master processor with status STUCK.

Step 6 else if $N_{update\ freq}$ number of Temperature-runs since last update, report solution to Master processor with status UPDATE.

Step 7 wait and receive solution from Master processor.

Step 8 $T = \alpha T$

Step 9 go to Step 3.

Figure 3.2: ASMCPSA algorithm

alternative solution to work on from the master processor. This process provides the MC with an opportunity to escape being trapped at its current solution and to explore a different part of the search space. Often, the MC is able to find an improvement to the best solution held by the master processor and hence is able to make a useful contribution to the search process by improving on the global best solution.

Thus, when Stuck, the slave processor reports its current solution to the master processor along with a status flag indicating that it is stuck. The master processor updates its global best solution according to the reported solution and returns a copy of the best solution to the slave processor. Each slave processor is allowed only an $N_{stuck\ limit}$ number of such requests, after which the MC follows the normal path of termination when deemed frozen if no transitions are made in N_{frozen} number of subsequent temperature-runs. When an MC is Frozen, it reports the current solution to the master processor and exits. The master processor updates the global best solution and the algorithm is terminated when MCs on all the slave processors report their solution with a status indicating that they are Frozen.

Chapter 4

Implementation using *Vitri*

4.1 Introduction

The Distributed Simulated Annealing **(DSA)** algorithm described in this thesis has been implemented using a Distributed Computing Framework called *Vitri*. The original *Vitri* kernel was developed by Baugh [11] as a framework that can be easily extended to implement many distributed computing applications. The framework has received numerous contributions from various graduate students who have added various optimization tools and decision support systems. This chapter discusses how the distributed computing kernel in *Vitri* has been used to implement the DSA algorithm.

4.2 Distributed Computing Module in *Vitri*

All distributed and parallel algorithms essentially split the work to be done as independent subtasks **(Tasks)**, which can be evaluated on various processors available in the distributed/parallel environment being used. Each of the Tasks that need to be performed has data and some operations associated with the data. Thus in order to perform distributed computing there is a necessity for some form of communication between various participating processors in a distributed computing environment to pass data and/or operations (Tasks) to be performed on the data between various processors.

Vitri provides a simple yet powerful framework which facilitates the encapsulation of these Tasks as independent Java objects which can be sent to the processors where they can be processed. *Vitri* also implements a "pool of tasks", illustrated in Figure 4.1, which takes care of efficiently scheduling the subtasks among various participating processors. Apart from this *Vitri* also provides features for the control of tasks as well as communication protocols.

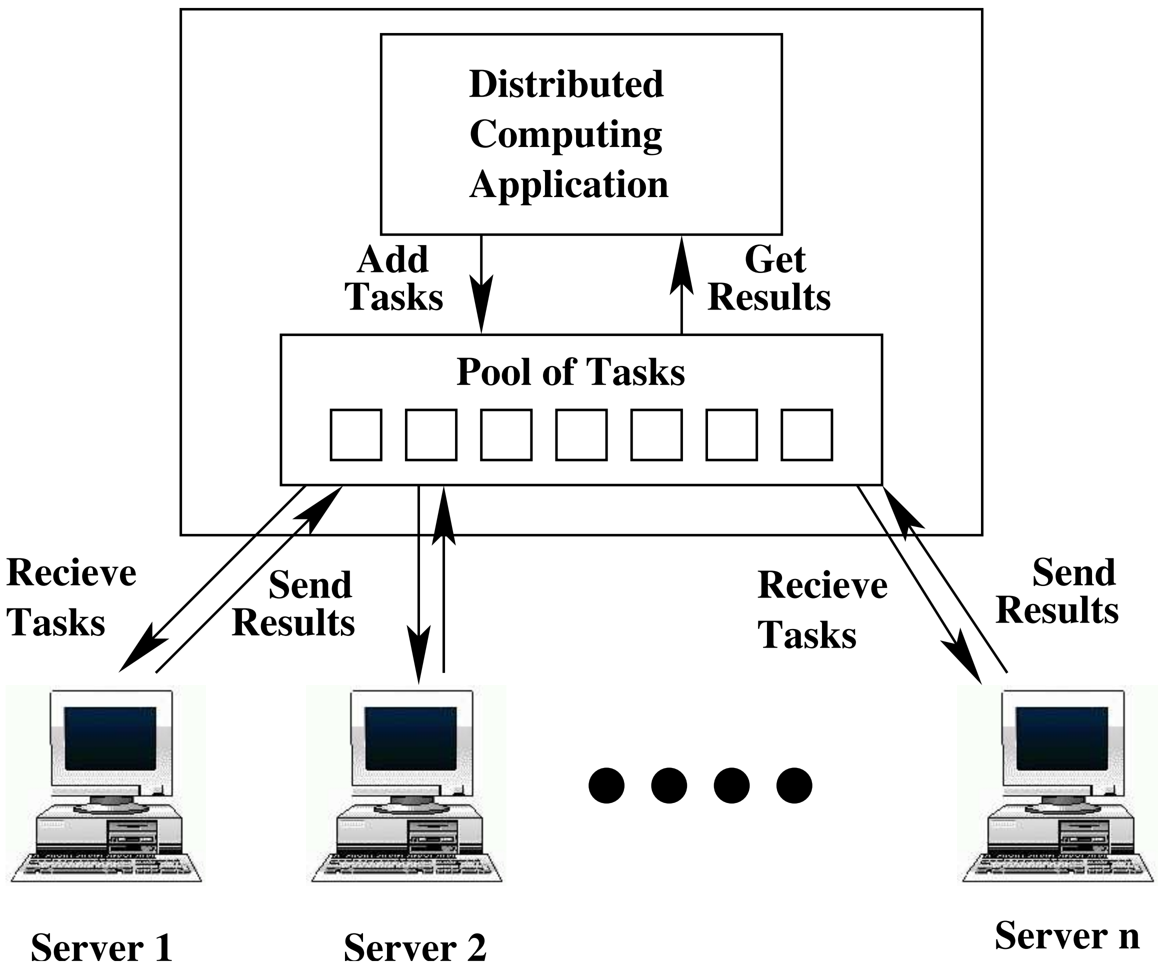

Figure 4.1: Pool of Tasks

Thus, any application which needs to use distributed computing can do so by first identifying the subtask (Tasks) that need to be performed. These Tasks are "submitted" to the pool of tasks, which is responsible for evaluating them. The pool coordinates the communication with various slave processors in the distributed computing environment and is in charge of forwarding the tasks to the slave processors. The slave processor (Servers), upon performing the task, report the results found to the pool. The pool returns a `TaskHandle` for each of the tasks submitted, which can be used by the application to track the status (whether it has been evaluated) of the task and to obtain the output associated with it.

4.3 Architecture of *Vitri*

4.3.1 Programming Environment

Vitri as well as the DSA algorithm have been implemented entirely using Java. Java being a strong object oriented language provides very good features to develop a simple, flexible framework. Being architecture neutral, Java makes using the framework over a heterogenous network computers very easy. *Vitri* and the DSA algorithm have been used on machines of various architectures with various operating systems, vis-a-vis

- Sun Ultra Sparc 5 workstations running Sun-Solaris
- Intel Pentium 3 processors running Red Hat Linux 7.3
- Intel Pentium 3 processors running Windows 2000

The communication between various processors is done via Java Sockets using TCP/IP over Ethernet.

4.3.2 Class Structure

The distributed computing kernel of *Vitri*, as shown in Figure 4.2, consists of four classes, vis-a-vis

- **Task**

  As described earlier the distributed computing application is responsible for identifying the subtasks that can be performed on a slave processor. The Java Interface `Task` provides a template which can be used to encapsulate work to be delegated to a slave processor. The `Task` interface defines a `runTask()` method which can be implemented to execute the designated work. The program running on a slave processor receives an object that implements this interface and executes the method `runTask()`, thus executing the "task."
- **Server**

  As described earlier the Distributed Computing Environment in *Vitri* consists

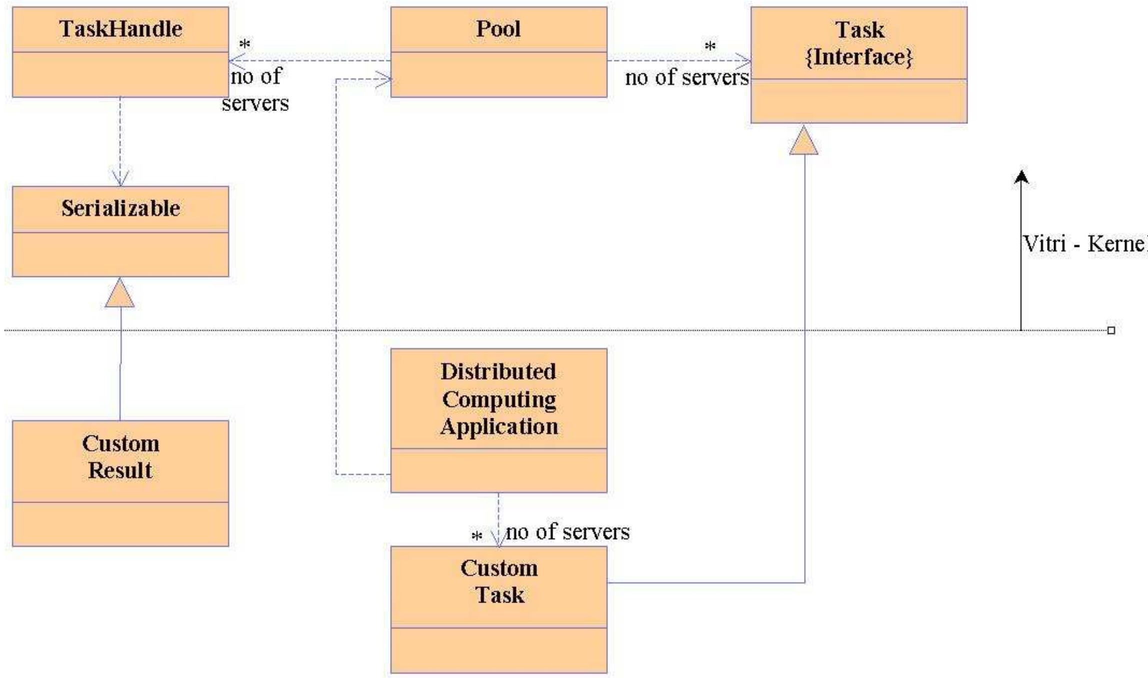

Figure 4.2: Class Diagram of Distributed Computing Kernel

of a master processor (or computer) where the distributed computing application is initialized and a set of willing slave processors which execute the tasks assigned to them. The class `Server` implements the responsibilities of slave processors which are to receive objects that implement the `Task` interface, to call the `runTask()` method of the object received, and finally to send the send the result object returned by the `runTask()` method to the master processor.

- **Pool**

  The Pool class manages connections to the various servers, accepts tasks, assigns tasks to various Servers (or computers running the Server program) and finally provides means for of reporting the results of a task to the appropriate process. When the pool is initialized, a `ManagerThread` is started which opens a ServerSocket on a specified port and listens to the incoming connections from Servers. When a connections request is received, a separate ConnectThread is spawned which opens a socket and stays in communication with the Server. The Pool class acts as a pool of tasks to which `Tasks` can be submitted for evaluation and from which results of executing the `Tasks` can obtained upon their evaluation. For every `Task` submitted to the pool, a `TaskHandle` is returned which can be used to monitor the status of the task and also to access the result upon its evaluation. The pool also acts as a central point from which all the Servers running on different machines can be terminated once they are no longer needed.

- **TaskHandle** The `TaskHandle`, as the name suggests, provides a handle on the Task submitted and can be used to monitor the status of the Task being evaluated. Upon the completion of the corresponding Task, the result field of the `TaskHandle` is set to the result of the execution of the Task and all Threads waiting on the `TaskHandle` are notified. Thus the result of executing a task can be accessed via its `TaskHandle`.
- **TerminateServer** The TerminateServer class implements the Task interface and is used as a marker class to signal the servers to shutdown. The Pool class sends this Task to the Server when the distributed computing environment has to be terminated.

4.3.3 Salient Features

Vitri provides a number of features which make it very suitable for the purpose of developing this DSA algorithm. Some of the important salient features are as follows

- **Heterogenous, Extensible and Flexible**

  Vitri has been implemented entirely in Java using TCP/IP sockets for interprocess communication. As Java is a platform neutral language and TCP/IP can be used for communication across most networks, *Vitri* can be used across platforms running any operating system as long as the appropriate JVM(Java Virtual Machine) is available. *Vitri* has been developed using principles of object oriented design as an extensible, flexible framework and allows for easy integration into any distributed computing application.
- **Load Balancing**

  In *Vitri* , the "pull" strategy has been used instead of "push" strategy for load balancing, where the Server program continually requests the master processor for Tasks to perform until it receives an explicit signal to terminate. Therefore, no slave processor stays idle as long as there are Tasks to be done, hence automatic load balancing is achieved.

- **Fault Tolerence**

  Processors can be added to the distributed computing environment at any point of time by starting the Server program on any new machine. Every time a new server program is started, the `ManagerThread` in the Pool class handles the connection and can allocate tasks to the new slave processor. Also, if the connection with any of the slave processors is disrupted, the corresponding task is again placed in the task pool, thus ensuring that algorithm proceeds unhindered.

All these features make *Vitri* an ideal choice for implementing the DSA. The subsequent section will provide more information on the implementation of the DSA algorithm using *Vitri* .

4.4 DSA implementation

The DSA algorithm described in the previous chapters describes how to decompose a simulated annealing algorithm into a number of parallel concurrent Markov chains, each of which can be executed independently on a remote machine. In order to implement this DSA algorithm, we need a framework which can be used to execute each of these Markov chains on remote machine, for which the distributed computing capabilities provided by *Vitri* were used.

The DSA algorithm has been implemented as an object oriented framework called the DSA framework. This framework abstracts the problem description from the DSA algorithm, implements all the activities to be performed as part of the DSA algorithm, and provides inteface classes using which most problems can be modeled to use the DSA algorithm. The DSA framework has been implemented entirely in Java and primarily consists of the following 4 classes,

- `DistributedSA`
- `SimulatedAnnealing`
- `Matter`

- `SATemperatureModule`

Each of these classes implements a portion of the DSA algorithm. For example,

- The `DistributedSA` class implements all the activities that need to be performed by the master processor,
- The `SimulatedAnnealing` class implements the `Task` interface to perform all the activities that need to performed by a Markov chain,
- The `Matter` class provides an abstraction to the problem being solved
- The `SATemperatureModule` class provides static methods which can be used to determine the initial temperature with which to start the DSA.

Apart from these classes, various other classes were implemented to model the problems relating pipe support location optimization, truss optimization and knapsack problem.

The parameters governing the DSA algorithm are specified in a file called DistSa_info.txt. This file provides access to a number of parameters governing the execution of the DSA framework and can be edited by users to customize the framework to meet their requirements.

The DSA framework utilizes the distributed computing capabilities of *Vitri* by using the `Pool` class to coordinate the execution of the MCs. Figure 4.3 shows the class structure of the DSA framework as well as the interaction with *Vitri* .

4.4.1 Framework Details

As described in the previous section, the DSA framework consists of 4 classes, each of which implements a portion of the DSA algorithm. The following is a detailed discussion on the scope and implementation details of the classes constituting the DSA framework.

Matter class

The `Matter` class is an abstract class meant to be the place holder for the solution representation and operators to perform transitions, evaluation of this solution representation. The DSA framework works with this abstraction of the problem and implements the rest of the DSA algorithm. Any problem which is to be solved using the DSA framework is to be formulated as a class which extends the Matter class implementing the transition

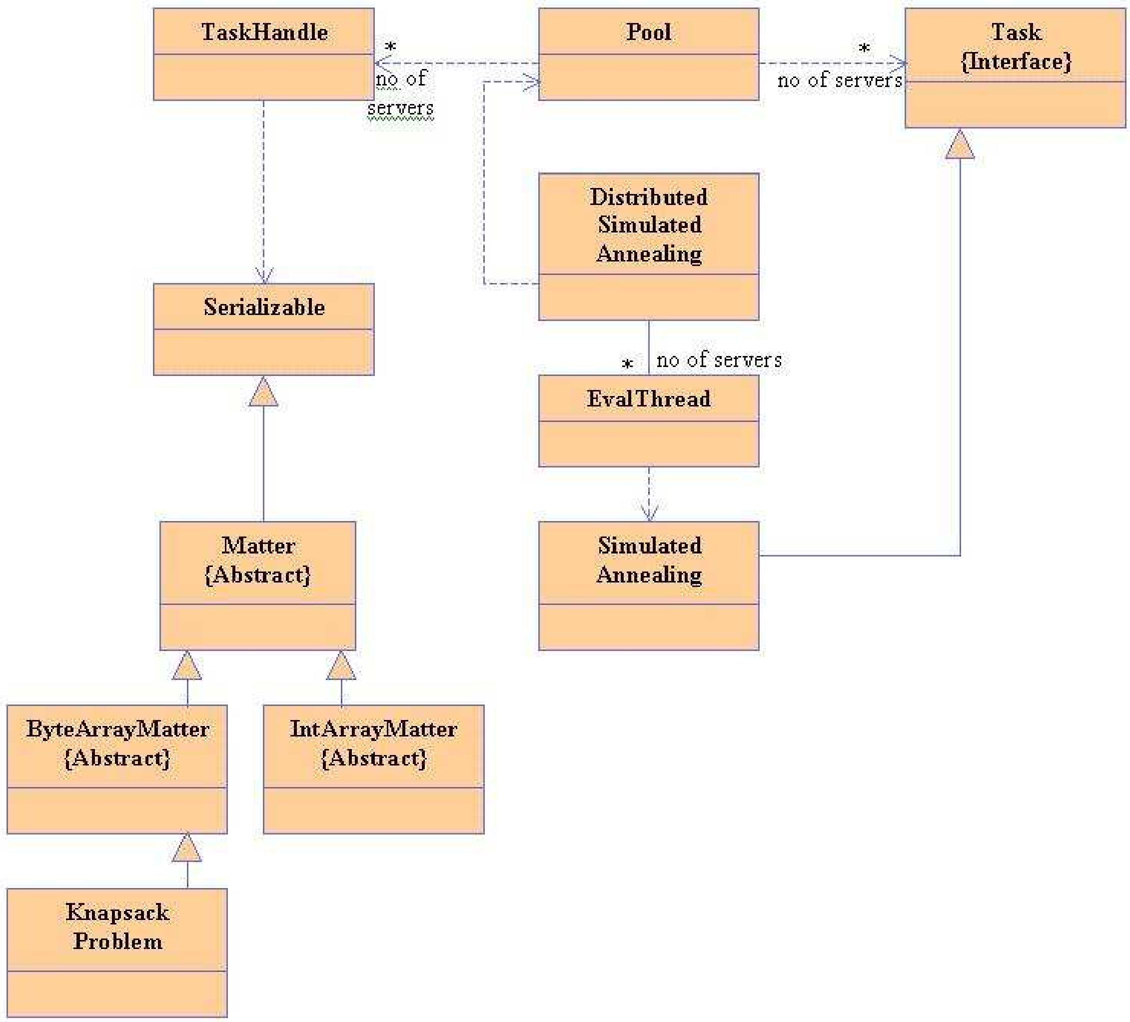

Figure 4.3: DSA framework Class Diagram

rules and evaluation rules for the solution representation of the particular problem. These rules are defined by implementing the abstract methods doTansition(), undoTransition() and analyze() declared in the abstract `Matter` class.

Each instance of the `SimulatedAnnealing` class in the DSA framework carries an object that extends the `Matter` class, and performs the operations of an MC on this object. As can be seen, the instance of the matter class maintains the solution representation and performs operations on the solution as and when instructed by the MC. The DSA algorithm requires communicating the solution held by an MC to the master processor at frequent intervals. At such times the matter held by the MC is sent to the master processor along with variables describing the state of the MC (the temperature, number of requests sent to the master processor, what status the matter is being reported with, etc.). Thus the `Matter` class has to play the role of the Result object returned by a Task class of the *Vitri* framework. Therefore the `Matter` class implements the `Serializable` interface as required by *Vitri* to implement the `Result` object (refer to Figure 4.3).

SimulatedAnnealing (SA) Class

As described in the previous chapter, the DSA algorithm attempts to split a sequential SA into a number of concurrent Markov Chain (MCs) which are executed by different processors. Each of these MCs corresponds to a Task as defined by *Vitri* because each MC consists essentially of data (the solution representation for the problem) and operations which need to be performed on the data.

The class `SimulatedAnnealing` provides all the functionality required to perform the activities of an MC by implementing the interface `Task`. Thus in order to perform evaluation of an MC on a slave processor, all that needs to be done is to create an appropriate instance of the `SimulatedAnnealing` class and submit it to a pool of tasks, where upon *Vitri* takes care of scheduling the task to be performed on a remote machine and passing the result obtained to the DSA framework.

As discussed earlier, an MC can be viewed as a Sequential Simulated Annealing algorithm with a reduced number of attempted transitions at each temperature (refer to Section 3.2.1). Therefore, as the name suggests, the `SimulatedAnnealing` class essentially provides a flexible sequential simulated annealing algorithm, for which the parameters can be controlled easily. Thus the `SimulatedAnnealing` class can be used as a sequential SA by instantiating an SA with appropriate values for various SA parameters. When initializing the various MCs, the DSA framework essentially creates a number of instances of the `SimulatedAnnealing` class each with a reduced number of attempted transition. These `SimulatedAnnealing` objects are then disbursed to various machines and results collated using *Vitri* .

Each instance of `SimulatedAnnealing` carries an instance of the `Matter` class, called the **incumbent**, which carries the solution. The `SimulatedAnnealing` instance calls on the methods `dotransition()` and `undoTransition()` implemented in the incumbent object to perform the various steps of the simulated annealing algorithm. When a solution is to communicated to the master processor, the `SimulatedAnnealing` returns the incumbent matter to the Server program, which in turn sends it to the master processor.

The DSA algorithm calls for communication between an MC and the master processor either to send an update, request a solution or to report a final solution.

The server program in *Vitri* communicates with a pool of tasks by sending the result of executing the `Task` only when its execution is complete. Therefore, when the `SimulatedAnnealing` class has to report a solution to the master processor, the solution can be sent to the master processor only by exiting the MC (i.e., the Task). When a solution has to be reported to the master processor, the `SimulatedAnnealing` class saves its state parameters in the incumbent being sent to the master processor and exits. It is expected that the master processor would analyze the result received from the slave processor and initialize another instance of the `SimulatedAnnealing` class with the state parameters set to those carried by the result. Thus, once the `SimulatedAnnealing` object is scheduled and executed on a slave processor by the `Server` program, the MC continues from where it was last interrupted.

The `SimulatedAnnealing` class can be initialized to act either as sequential simulated annealing algorithm or as an MC as needed. The input file DistSa_info.txt can be used to customize the parameters describing the behavior of the MC or the sequential SA by defining parameters , such as temperature, number of attempted transitions at each temperature and initial temperature for the annealing scheme.

DistributedSA

The `DistributedSA` class constitutes the most important part of the DSA framework and implements a Distributed computing application (see Figure 4.2) using the pool of tasks provided in *Vitri* . This class executes all the activities to be performed on the master processor of the DSA algorithm and acts as the control center for the DSA framework.

When an instance of `DistributedSA` is executed, the file DistSa_info.txt is examined for user specified information. After gathering the information, a pool of Tasks is initialized, a user specified number of instances of the class `SimulatedAnnealing` are initialized. For each instance of the `SimulatedAnnealing` class, the `DistributedSA` spawns a Thread which is responsible for submitting the `SimulatedAnnealing` object to the pool of tasks and to invoke the `update()` method of the `DistributedSA` class with the result of execution of its assigned `SimulatedAnnealing` instance. Thus, the process of submitting MCs to the pool of Tasks and obtaining the result of execution of the MC is automated. The class `EvalThread`, which is a class internal to `DistributedSA`, implements these threads.

The `DistributedSA` class maintains the best solution reported by the currently executing MCs. When a result is received from a slave processor, the reported solution is compared to the current best solution held by the `DistributedSA` and overrides the best solution with the new solution if the latter is better. Each solution reported also has a status associated with it that determines the actions taken. If the solution is returned with a status of

- UPDATE, STUCK a new instance of `SimulatedAnnealing` class is initialized with the best solution carried by the `DistributedSA` class as the Incumbent and its state parameters set to those carried by the reported matter.
- FROZEN, no further instance of the `SimulatedAnnealing` class is initialized and the number of completed MC is incremented.

As described earlier, the `DistributedSA` class initializes a user defined number of instances of `SimulatedAnnealing` class. Every time a `SimulatedAnnealing` object completes execution, i.e. reported as frozen, the MC is considered to be completed. When all the initialized MCs are completed, the best solution is reported as the optimal solution found and the framework is stopped by calling for termination of the pool of tasks.

SATemperatureModule

The initial temperature to be used for a the DSA framework can be specified by the user. The initial temperature to be used varies from problem to problem depending on various factors like solution representation, neighborhood size, nature of the problem among others. The `Matter` class defines an abstract method `getInitTemperature()`, which is to be implemented by any problem for which one would like to the DSA framework. Thus, the users can implement their own code to find the initial temperature.

The `SATemperatureModule` class has been developed as a class that implements the method outlined in Section 2.4 to find the initial temperature for simulated annealing algorithm. Two independent methods have been implemented that can be used to determine the initial temperature either on a single machine or on multiple machines. The user can execute these methods to determine the initial temperature to be used for the particular problem.

Other Classes

Apart from these classes, a few additional ones were implemented that extend some of the classes of the framework and provide functionality that can be useful in modeling certain classes of problems. For example, the `ByteArrayMatter` and `IntArrayMatter` classes extend the `Matter` class and can be extended to model problems that need arrays of variable of type Byte or int, respectively, to represent the solution. These classes do not implement the `analyze()` method, thus allowing for classes which extend this class to implement this method as needed.

4.5 Summary

The DSA algorithm has been implemented as an object oriented distributed computing application using *Vitri* . The framework is extensible and defines easy to use classes and interfaces to implement most optimization problems. End users are required to extend the class `Matter` to implement their problems. Users can also customize the various parameters used by the framework to suit their needs. Appendix A gives a detailed explanation on how to use the DSA framework to solve a user defined problem and also shows a sample DistSa_info.txt file.

Chapter 5

Porting Vitri to a Parallel Platform

Java has emerged as a popular object oriented language in recent years. Its object oriented nature, extensive API, strong type casting and ease of use are among many features which have made it popular. The suitability and use of Java for scientific computing, especially on massively parallel architectures, has been widely debated and actively researched in the scientific circles.

Several Java interfaces, like mpiJava [1], JOMP [3], HP-Java [4], PJMPI [17] and JOPI [18] to name some, are available to the widely used MPI message passing interface. These APIs provide Java like bindings to MPI calls which can be used for communication on a parallel machine. Thus, enabling development of object oriented applications capable of executing on parallel architectures.

As has been discussed in previous chapters, the DSA framework is a simple, easy to use framework for using the DSA algorithm to solve many kind of engineering optimization problems. Apart from the DSA framework, a number of other engineering optimization tools have been implemented that take advantage of the distributed computing capabilities of *Vitri* . However, it has been designed to run only on a cluster of networked desktop computers. The ability to use optimization software developed for a cluster environment on a parallel machine would be of great practical value.

Several of the Java interfaces to MPI mentioned earlier are ideal for developing object oriented frameworks to perform parallel computing. Therefore, to facilitate the use of codes developed for a distributed cluster to run on a parallel machine, the *Vitri* framework was ported to a parallel platform using the mpiJava API [1].

The DSA framework discussed in this thesis was initially implemented using *Vitri* version 1.0 and was later adapted to work with *Vitri* version 2.0, which provides a much simpler interface to its distributed computing features. Although only version 1.0 was ported to a parallel platform using mpiJava, version 2.0 can be ported similarly with ease. This chapter provides the implementation details of the ported version of *Vitri* called **Vitri-MPI**, and the results obtained on solving a knapsack problem on an IBM-p690 machine using this ported framework.

5.1 *Vitri* 1.0

Figure 5.1 shows a class diagram of Vitri 1.0, where the class structure is very similar to the class structure of version 2.0 (refer to Figure 4.2). It can be seen that the abstract classes `Program`, `CustomProgram` and `Client` constitute the distributed computing application part of Figure 4.2, and the class `Job` is analogous to class `TaskHandle` and acts as a handle on the various tasks added to the pool.

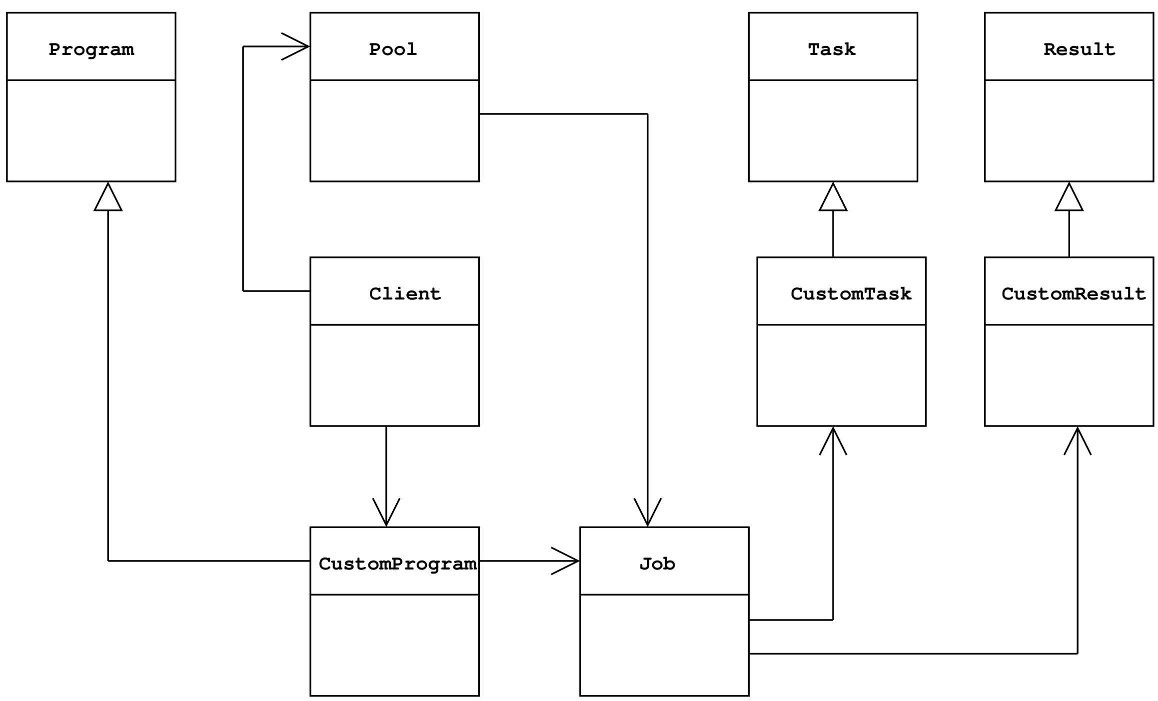

Figure 5.1: Vitri 1.0 Class Diagram

The abstract class `Program` is an abstraction for the distributed computing application, which uses the instance of the pool in the Client class to coordinate communication with slave processors. The class CustomProgram is a concrete class that implements the particular distributed computing application, creating Tasks, wrapping a Job (which acts as the TaskHandle) around these Tasks and submitting to the pool and collating the results as and when they arrive.

5.2 Porting

Vitri uses TCP/IP sockets for communication between various processors and uses IP addresses to identify and assign tasks to various slave processors. However, on a parallel architecture, a number of processors consistute an "MPI_COMM_WORLD" which is an environment of participating parallel processors assigned to a single run of **Vitri-MPI**. Each processor is identified by its "rank" in the environment and communication between various processors is performed using the message passing paradigm. mpiJava provides a Java API for the underlying MPI message passing libraries available on the parallel machine which is used by Vitri-MPI to obtain the "MPI_COMM_WORLD" and use the MPI calls for sending Tasks and obtaining results from the slave processors. The following are some of the important classes and their functionality in Vitri-MPI.

Client

The Client program performs all the activities that need to be performed by the master processor. The Vitri-MPI framework is started by executing the Client program with the total number of desired processors. The Client program obtains a handle on the "MPI_COMM_WORLD", obtains the total number of processors in this run, starts the activities of the master processor on the processor of rank 0 and starts the Server program on all the remaining processors.

CustomProgram

This class actually implements the parallel computing application that intends to use Vitri-MPI. For example, an equivalent of the `DistributedSA`class (called `SAProgram`) is implemented for this class, which essentially creates the various tasks (MCs) and uses the Pool for performing the parallel computation. This class is identical both on the parallel and distributed computing environments, thus a distributed computing application developed using *Vitri* can be used as is on Vitri-MPI.

Pool of Tasks

The `Pool` class is the hub of all the communication and manages the communication with all the slave processors. In a distributed computing environment, the Pool communicates with slave processor by using TCP/IP over an Ethernet LAN. This is replaced by using MPI calls to send data (essentially objects) from the Pool to servers and vice-versa. The Pool class obtains information about the "MPI_COMM_WORLD" from mpiJava calls and uses the information on the participating slave processors. A separate ConnectThread is spawned to handle communication with each slave processor. When the CustomProgram has completed its work, the control passes to the Client program which calls on the shutdown() method in the `Pool` class. The Pool class terminates the Servers by sending a "null" object, after which the servers and the Client program exit.

CustomSerializable

mpiJava provides MPI-like calls to send objects from one processor to another. However, costly startup and latency overheads could be reduced if number of objects sent across the network is kept to a minimum. To facilitate this, a format for custom serialization has been defined, which defines what information from the object needs to be serialized and how the information should be received and used to recreate the object.

The CustomSerializable interface defines the methods `sendName()`, `writeSerializable()` and `readSerializable()`. Every class that is to be sent from one processor to another (essentially the Task class) is expected to implement this interface. The Vitri-MPI framework uses the methods defined in CustomSerializable to serialize and serialize an object.

When the pool is initialized, it gathers information about the participating processors in the “MPI_COMM_WORLD” and spawns a separate thread (ConnectThread) to handle communication with each of the slave processors. First, the name of the object is sent using `sendName()` method. The slave processor receives the name of the object and instantiates an object of the class with the name of received class. Then, the ConnectThread sends the user defined serialization procedure implemented in `writeSerializable()` to send the information and the slave processor uses the user defined deserialization procedure implemented in `readSerializable()` method of the incoming object.

MPIWrappers

This is a class of static methods which provide a useful abstraction to mpiJava calls for message passing. These can be used by the user to conveniently serialize and deserialize the objects to be sent from Pool to servers and vice-versa.

Thus, in order to port *Vitri* , the `Pool` class was reimplemented to handle the communication using MPI, the Client and Server classes were modified to use “MPI_COMM_WORLD” for identifying and using the parallel environment.

A distributed computing application (`CustomProgram` class) developed using *Vitri* can be used as-is on Vitri-MPI. The CustomTask class needs to be adapted to work with Vitri-MPI by implementing the CustomSerializable interface. Implementing these methods in this interface is very easy as the MPIWrappers class provides useful methods for serializing and sending objects on the parallel platform. Thus it can be seen that Vitri-MPI provides a convenient easy to use framework on the parallel platform and can be used to execute the codes developed using *Vitri* for a distributed computing environment on a parallel machine.

5.3 Test case

In order to examine the performance of the Vitri-MPI framework, the 40-variable knapsack problem, with 40 ms sleep introduced in each function evaluation, was solved using an implementation of the DSA framework for *Vitri* version 1.0, on an IBM pSeries 690 machine at NCSA[1], Illinois. 3 cases were considered each with 40, 80 and 160 number of maximum attempted transitions at each temperature. Figures 5.2, 5.3, 5.4 present a comparison of solution quality vs execution for these cases on parallel and distributed computing environments.

A subsequent section, section 6.1 describes the knapsack problem formulation, its details and the parameters used to solve this. The same implementation used to solve knapsack problem on a Linux Cluster using *Vitri* was used on the parallel machine with minor modifications to allow custom serialization on the supercomputer at NCSA using Vitri-MPI. The quantities discussed in this section are averages of values obtained upon the solving each case using 4 different seeds.

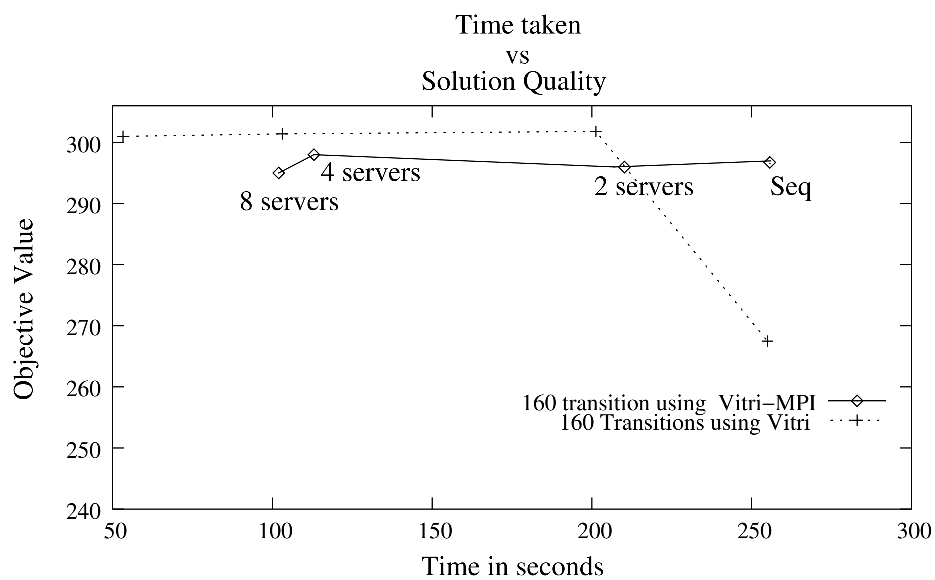

Figure 5.2: Solution Quality comparison of Distributed and Parallel runs

It can be seen that Vitri-MPI leads to a reduction in evaluation time and that the solution quality is comparable to that obtained using *Vitri* . The general behavior of the DSA algorithm can be seen here. However it can be seen that *Vitri* shows a more consistent reduction of execution time, over increasing number of servers, as compared to Vitri-MPI.

Figures 5.6, 5.5, 5.7 present the details of the speedup obtained for various cases. It can be seen that even though speedup increases with an increasing number of servers, in general

[1]The technical specification of this machine are available at `http://www.ncsa.uiuc.edu/ UserInfo/ Resources/ Hardware/ IBMp690/ TechSummary/`

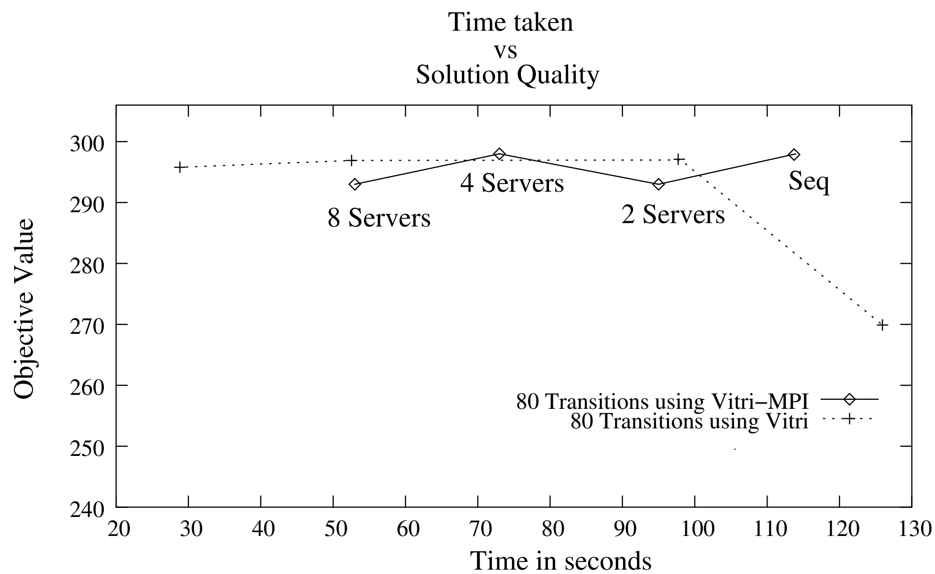

Figure 5.3: Solution Quality comparison of Distributed and Parallel runs

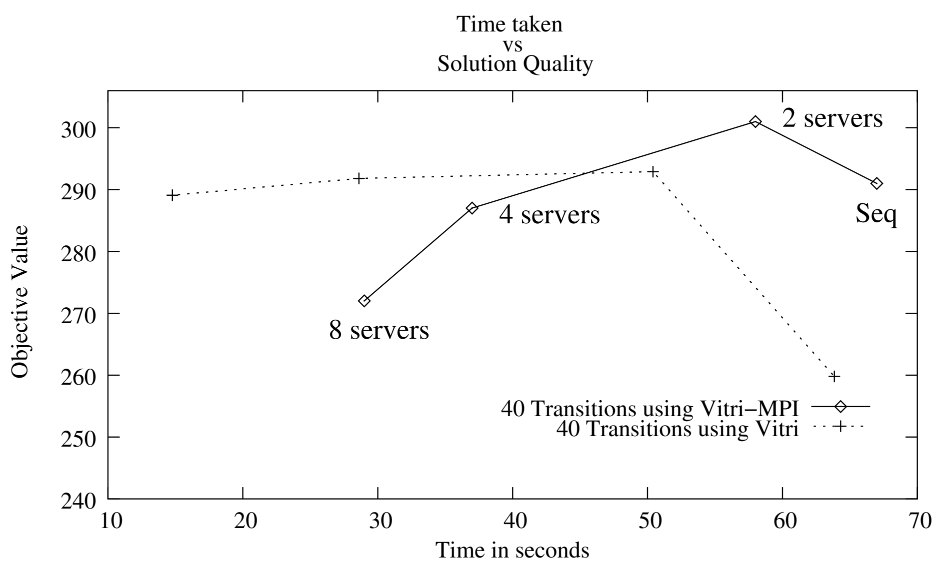

Figure 5.4: Solution Quality comparison of Distributed and Parallel runs

the speedup obtained for any particular case using *Vitri* is better than those for Vitri-MPI.

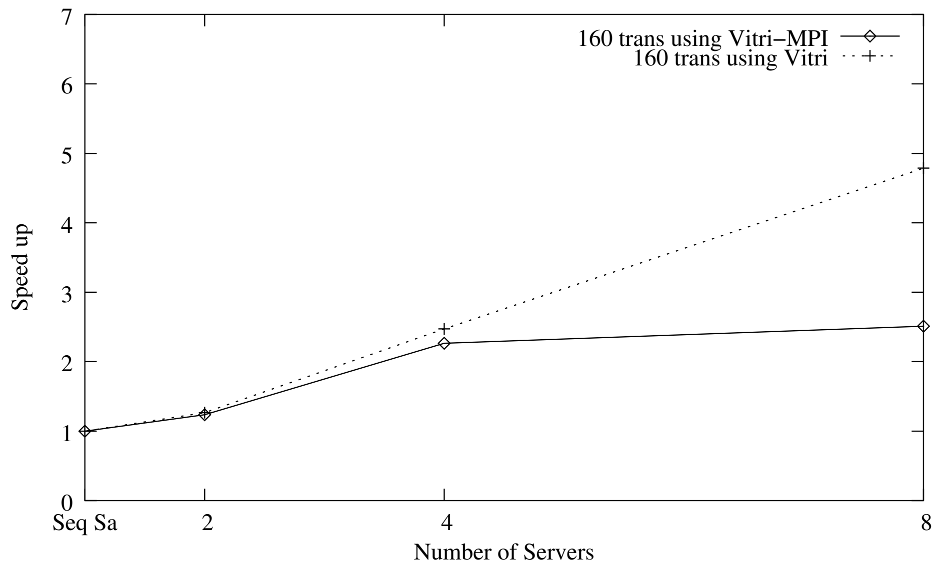

Figure 5.5: Speedup comparison of Distributed and Parallel runs

This speedup behavior translates to the efficiency of Vitri-MPI as well. Figure 5.8 presents the speedups for various cases using *Vitri* and Vitri-MPI frameworks. It can be seen that Vitri-MPI shows efficiency comparable to *Vitri* for the two server case. However, it can be seen that efficiency decreases with an increasing number of servers. This implies that the framework is not as scalable on the parallel programming environment as *Vitri* is on the distributed computing environment.

Thus it can be seen that Vitri-MPI brings the benefits of *Vitri* in reducing the computational time by distributing the work among a number of servers. The DSA algorithm was executed using Vitri-MPI to solve the knapsack problem and comparison against the results obtained for solving the same problem using *Vitri* shows that though the general behavior of the DSA algorithm are reproduced on the parallel platform, the scalability and the speedups observed on the parallel platform are not as good those observed on the distributed computing cluster.

It is well known that Java compilers have not been optimized to perform well on many parallel machines. This could be a major factor in Vitri-MPI not performing as well as *Vitri* . Even though the communication hardware on a parallel machine is much faster, at least

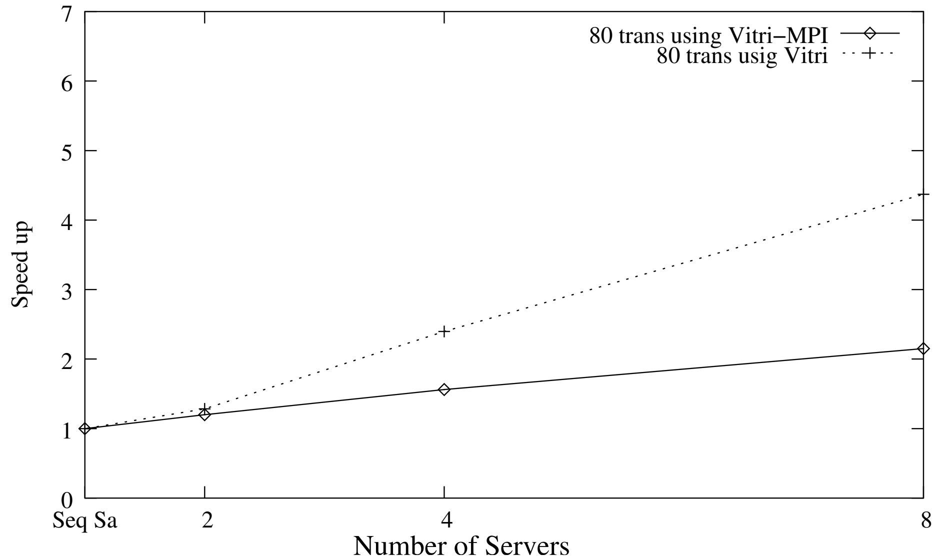

Figure 5.6: Speedup comparison of Distributed and Parallel runs

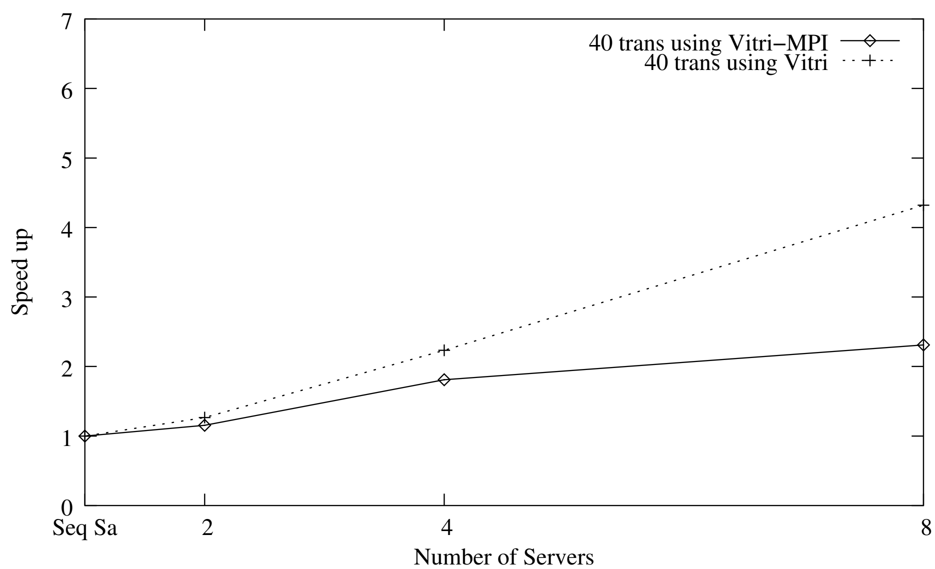

Figure 5.7: Speedup comparison of Distributed and Parallel runs

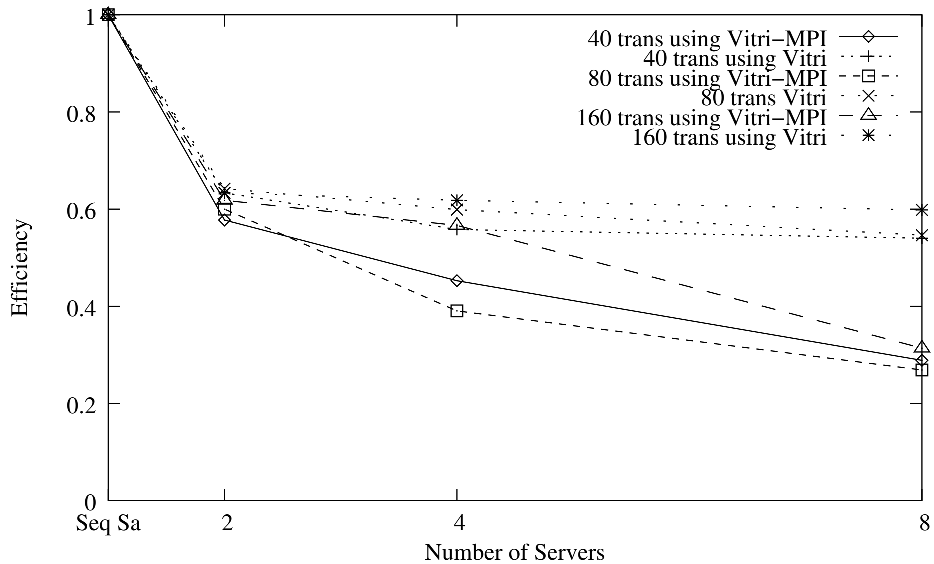

Figure 5.8: Efficiency comparison of Distributed and Parallel runs

20 times as fast as the Ethernet LAN of distributed computing environment, mpiJava may not be efficient in handling communication of complex objects. Also the process of sending and receiving objects involves substantial amount of serialization, which would again be affected by the performance of Java and mpiJava on the parallel architecture. These could be possible reasons for reduction in performance Vitri-MPI as compared to *Vitri* . Further developments in optimizing Java for parallel environments and mpiJava will be of great help in improving the performance Vitri-MPI.

Chapter 6

Results

The DSA algorithm has been used to solve benchmark heuristic optimization problems as well as real world applications. The well known knapsack problem was used as a benchmark problem to study and analyze the performance of the DSA algorithm. The performance of the algorithm was tested based on the following properties

- Solution quality vs time
- Speedup
- Efficiency
- Number of attempted and successful transitions.

Apart from the knapsack problem, the algorithm was used to solve a problem involving optimization of Support Location for a piping system subjected to Earthquake Loads.

All the executions of the algorithm were performed on a cluster of 9 desktop computers with Intel Pentium III processors running at 800 MHz, with 256 MB RAM running the Red Hat Linux 7.3 operating system. The computers were networked using a 100 Mbps Ethernet Local Area Network.

This chapter presents the results of the various experiments to solve the problem and presents an analysis of the performance of the DSA algorithm.

6.1 Knapsack problem

The knapsack problem is among one of the very well studied combinatorial optimization problems, which has been solved using a number of optimization techniques. In order to study the performance of the DSA algorithm, a binary variable knapsack problem of size 40 was formulated. The knapsack problem consists of a set of items, and sets of corresponding weights, benefits associated for each item. The objective is to select a number of items such that the sum of benefits of the items chosen is maximized while the sum of weights of the items chosen does not exceed a given limit i.e. maximum capacity.

For a knapsack problem of n items, the problem could be mathematically represented as,

b_i = Benefit associated with selecting item i,

w_i = Weight associated with selecting item i,

C_{max} = Maximum capacity of the knapsack

to find an assignment vector $X = (x_1, x_2, ..., x_n)$, where

$x_i \in \{0, 1\}$ and $x_i = 1$ iff item i is selected such that

$$\sum_{i=1}^{n} w_i.x_i \leq C_{max}$$

for which $f(X)$ is maximum, where

$$f(X) = \sum_{i=1}^{n} b_i.x_i$$

The DSA algorithm was tested using a 40 variable knapsack problem. The solution was represented as an array of "byte" variables which can take the values of either 0 or 1. Transitions from one solution to another during a simulated annealing process are performed by randomly selecting a location in the solution array and flipping the value held at that location. For this definition of solution transition, the neighborhood size is 40. Therefore the maximum attempted transitions at each temperature for an SSA algorithm was chosen as 40. The initial temperature for all the test cases performed was 20, which produced an initial acceptance rate of about 85-90% for this problem.

This problem was solved using a sequential run of the simulated annealing algorithm and DSA algorithm runs using 2, 4 and 8 slave processors. In the DSA algorithm, the maximum attempted transitions at each temperature by each Markov chain is a fraction of those performed for an SSA algorithm. In the case of 8 slave processors, the maximum attempted transitions on each Markov chain would be 1/8 th of those performed in a sequential algorithm, which is 5 maximum attempted transitions at each temperature. For problems with large neighborhood sizes, stepping down the number of attempted transitions at each temperature may not affect the convergence of the simulated annealing algorithm. However in this problem, each MC is performing only 5 transitions at each temperature. This leads to more chances of not attaining a quasi-equilibrium state at the end of each of temperature-run. Hence, the problem was attempted using 40, 80 and 160 attempted transitions at each temperature in order to consider cases with varying amounts of likelihood for non-attainment of quasi-equilibrium.

As has been discussed before, communication plays a major role in this algorithm. When solving fine grained problems in which each solution evaluation does not take much time, the communication time inherent to the DSA algorithm can be considerably higher than the computation time. Thus communication time becomes a big overhead which can make using the DSA algorithm unattractive. However when solving coarse grained problems with a large computational time for each solution evaluation, the communication time would be negligible compared to computational time, in which case the ability to distribute the evaluations among a number of processors reduces the overall execution time of the algorithm. In order to emulate a computationally intensive problem using the knapsack problem, a fixed amount of sleep time was introduced into each solution evaluation. Test cases were considered with a sleep time of 10 and 40 milliseconds respectively to compare the performance for problems of different computational intensities.

As the simulated annealing search process is essentially random in nature, results obtained can vary for each test run. In order to obtain a fair estimate for the performance of the algorithm, all the properties which were studied during the test cases were averages obtained from a number of runs with different seeds. All the properties used to analyze the performance of the knapsack problem were averages obtained from 10 different runs.

6.1.1 Solution Quality vs Time taken

The following figures present a comparison of solutions obtained and the time taken for various runs of the algorithm. Figure 6.1 shows the results obtained from running the algorithm for a knapsack problem with sleep time of 10 ms per evaluation. Each of the contours in this diagram has three points on it, each of which corresponds to cases where the maximum number of attempted transitions at a temperature was 40, 80 and 160 respectively. Figure 6.1 shows that for any number of attempted transitions, as compared to the sequential algorithm, the distributed algorithm is far superior in terms of the quality of the solution. It can be seen that there is some reduction in execution of time between the sequential execution and distributed execution using 2 servers. The execution time between distributed runs with various numbers of slave processors, reduces by a constant factor. Thus it can be seen that the DSA algorithm gives superior performance both in terms solution quality and the execution time compared to an SSA algorithm.

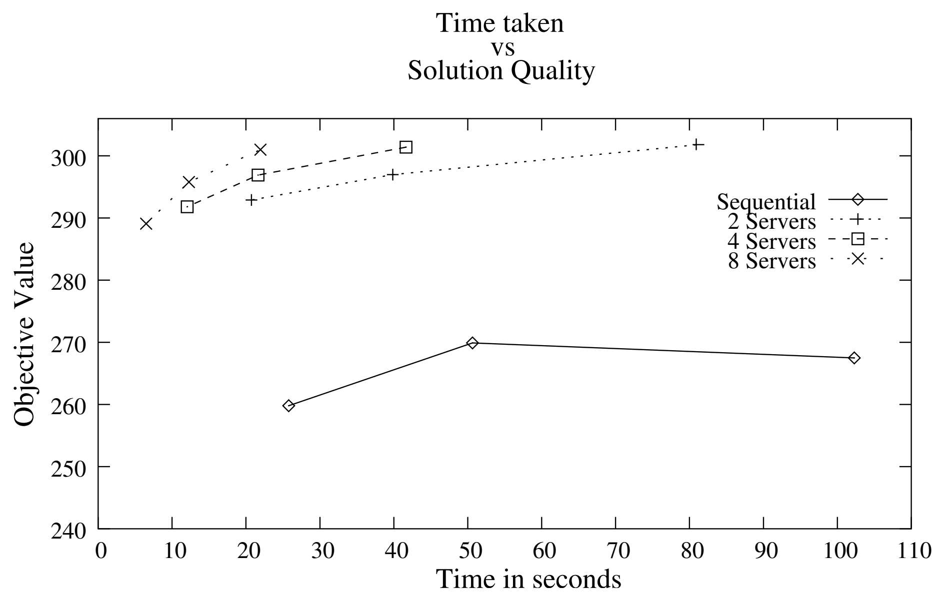

Figure 6.1: Time vs Cost

6.1.2 Speedup

Speedup is a measure of how fast the distributed algorithm finds a solution as compared to a sequential algorithm. The speedup obtained while using n slave processors can be expressed as

$$\tau = \frac{T_{seq}}{T_n}$$

where, T_{seq} is time taken by the SSA algorithm, T_n is time taken by the DSA algorithm using n servers.

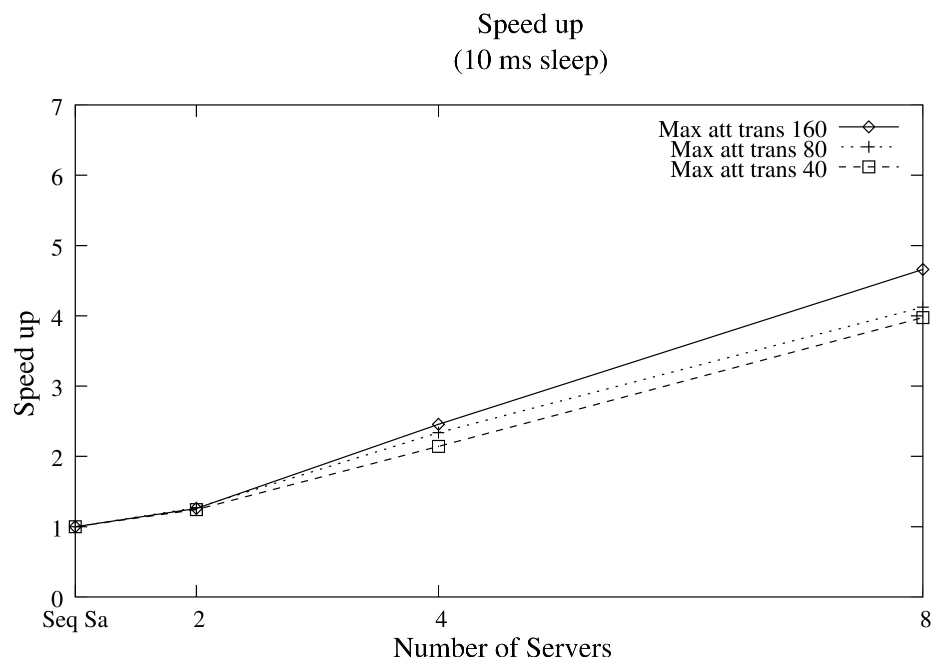

Figure 6.2: Speedup for Knapsack problem

Figures 6.2 and 6.3 present the speedup values obtained for various cases with a sleep of 10 ms and 40 ms respectively. It can be observed from the graphs that the speedup for both the cases is approximately linear for all cases. This implies that the execution time reduces almost linearly as the number of slave processors increases. For a knapsack problem with a sleep of 10 ms, a maximum speedup of 4.5 was observed, while in the case of 40 ms sleep, a maximum speedup of 5.2 was observed. The slope of the speedup curve is not high between the sequential and 2-slave processor case. However, it can be seen that the slope of the curve from 2 slave processors to 8 is about 2/3. Even though it would be highly desirable for the slope of speedup curve to be 1 for a distributed application, when number of attempted transitions in a distributed case is comparable to a sequential run of the algorithm, the communication overhead poses prevents distributed algorithm from showing the desired increase in speedup.

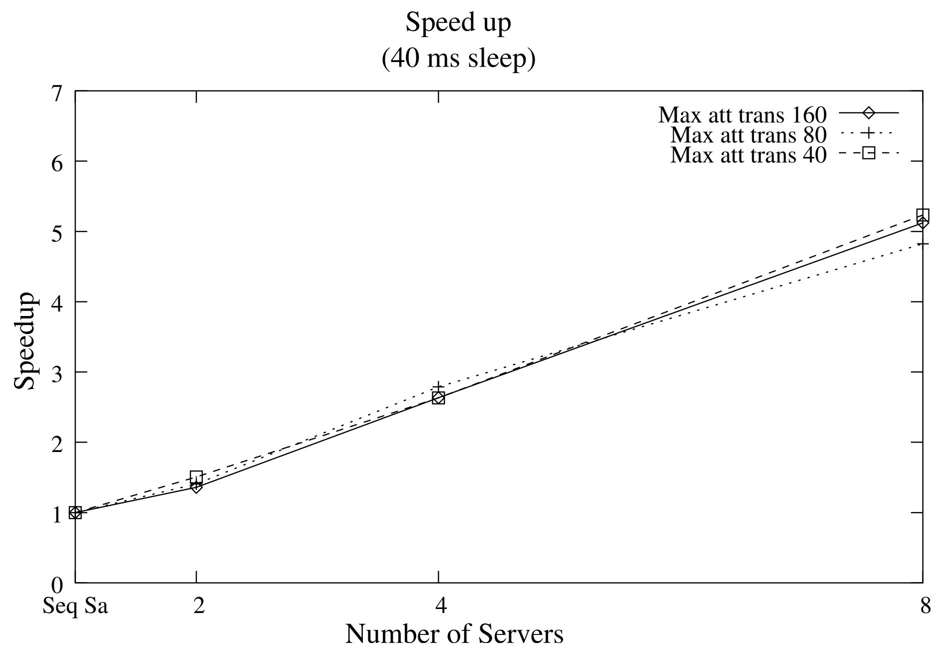

Figure 6.3: Speedup for Knapsack problem

However, the convergence characteristics of the DSA algorithm is affected due to the interactions between the MC. In a subsequent section, the effect of this interaction is discussed, where a superlinear speedup was observed.

Considering the fact that the number of transitions attempted by the MCs in this case is comparable or greater than those by SSA, the speedup curve indicates consistent performance over a varying number of slave processors.

6.1.3 Efficiency

Efficiency of the algorithm while using n servers can be defined as

$$\eta = \frac{\tau}{n}$$

where τ is the speedup obtained from using n servers.

Efficiency quantifies the scalability of the algorithm. If the efficiency of the algorithm remains constant with an increasing number of servers, it can be inferred that the algorithm scales well. However if the efficiency of the algorithm decreases with an increasing number of servers, it can be inferred that the algorithm does not scale well. Figures 6.4 and 6.5

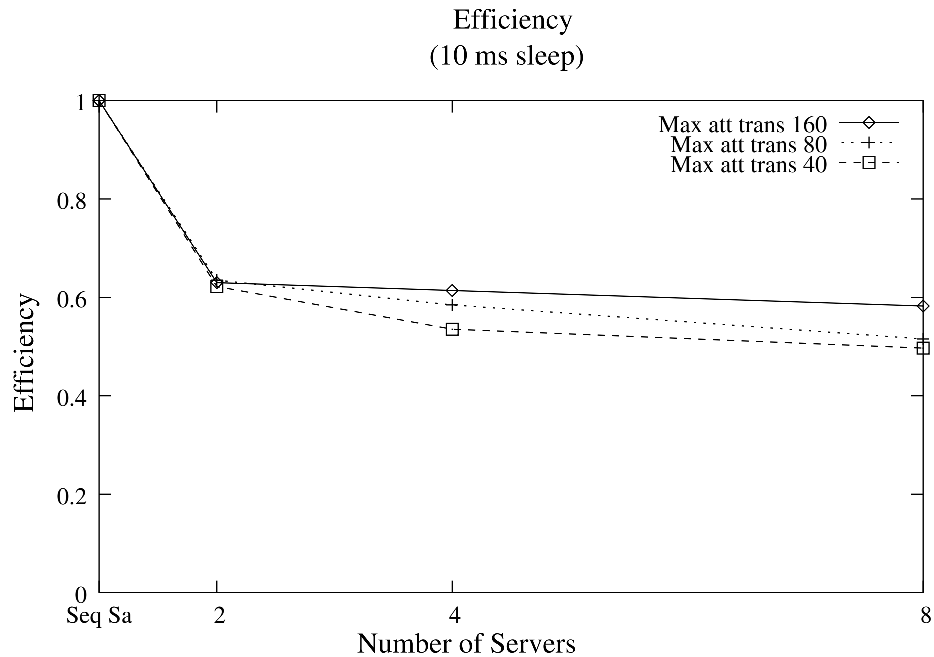

Figure 6.4: Efficiency for Knapsack problem

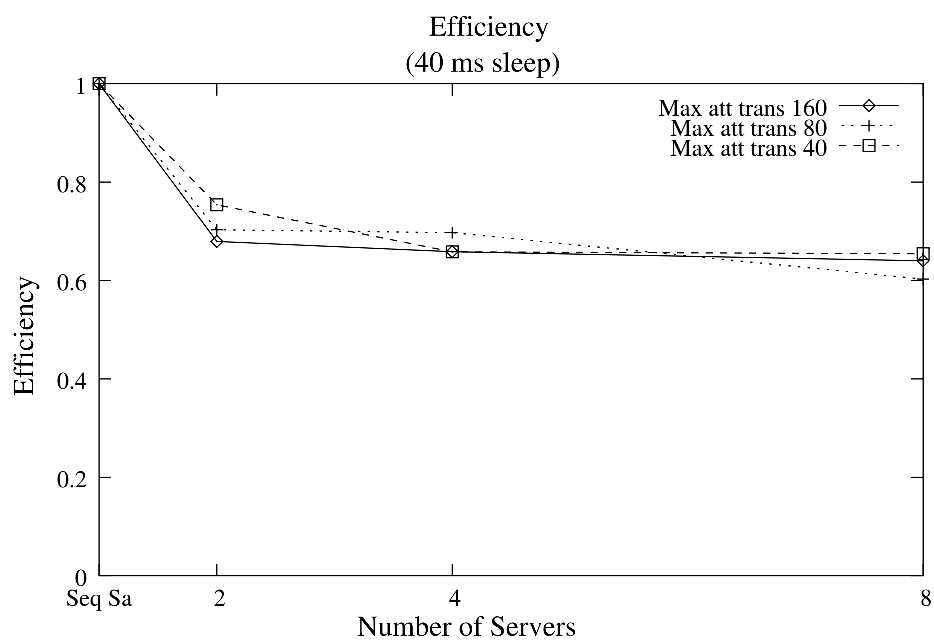

Figure 6.5: Efficiency for Knapsack problem

present the efficiency of the algorithm for 10 and 40 ms sleep, respectively. It can be seen that in most of the cases, the efficiency is approximately constant. In the case of the knapsack problem with a sleep of 10 ms, for various cases the efficiency is approximately 0.6. Similarly in the case of knapsack problem with a sleep of 40 ms, the efficiency is approximately 0.7. Thus the algorithm appears to be scalable until a size of 8.

6.1.4 Number of Attempted and successful Transitions

The DSA algorithm splits the work of a sequential algorithm among various processors. As the algorithm splits a sequential Markov chain into a number of co-operating Markov chains, the number of attempted and successful transitions performed by SSA algorithm is different from those by a Distributed algorithm. Figure 6.6 shows the total number of attempted transitions for various Maximum Attempted Transitions cases with a sleep of 10 ms. It can be observed that the DSA algorithm performs about 1.4 times as many attempted transitions as compared to those by the sequential algorithm. This ability to make a greater number of attempted transitions as compared to an SSA algorithm could be a reason for the better quality of solution using the DSA algorithm.

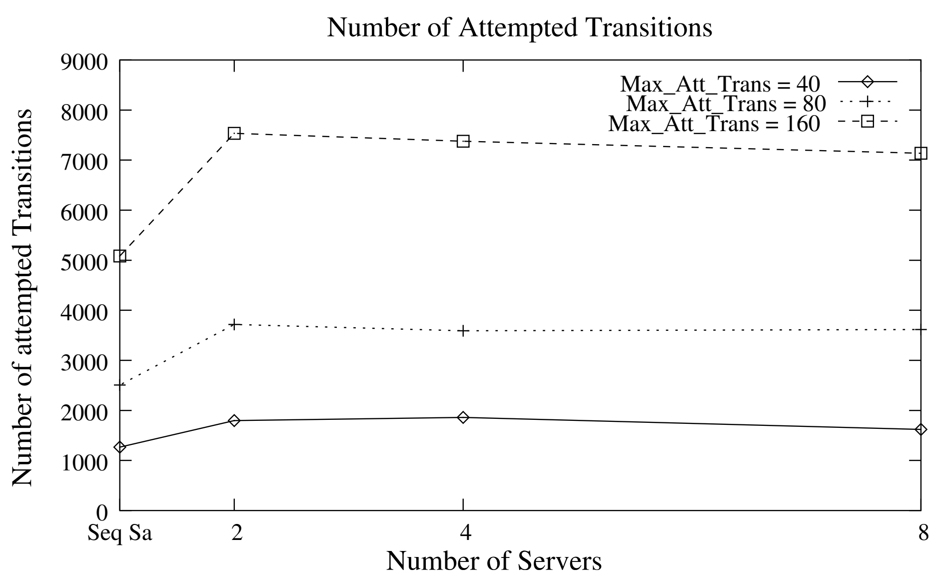

Figure 6.6: Number of Attempted Transitions

Similarly, Figure 6.7 provides information on the total number of successful transitions performed by the SSA as compared to the distributed algorithm with a sleep of 10 ms. It can be seen that the total number of successful transitions in Distributed Algorithm is about 1.2 times as that in sequential algorithm.

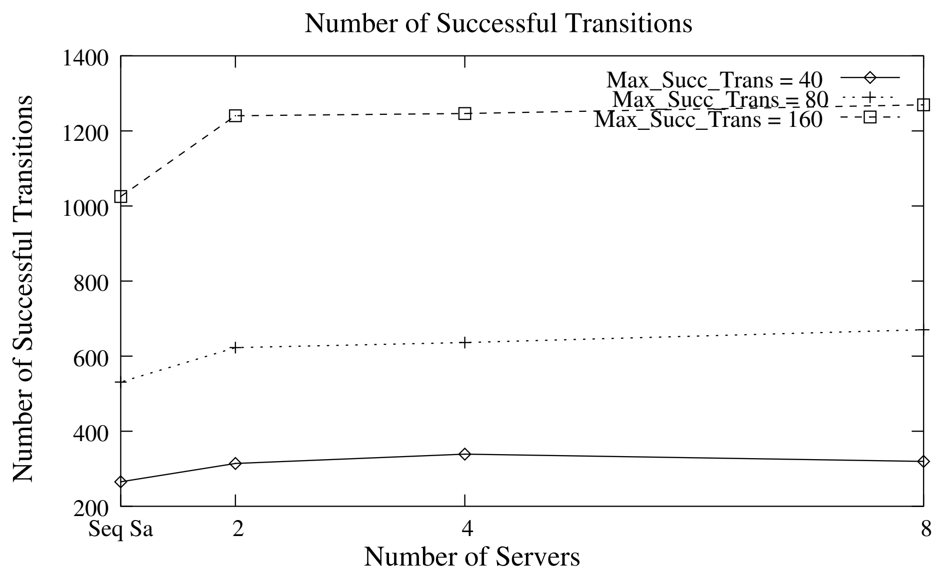

Figure 6.7: Number of Successful Transitions

Thus it can be seen that the DSA algorithm shows considerable improvement over the SSA algorithm both in terms of solution quality and in terms of execution time. The algorithm seems to show a slightly better speedup and efficiency for the 40 ms sleep case as compared to 10 ms case.

6.2 Optimization of Support Location in a Piping System

Design of piping systems is a heuristic process based on professional experience and judgment. However it is desirable to produce optimal designs using engineering knowledge and heuristic optimization techniques. Chiba *et al.* [5] have worked in optimization of pipe support locations using genetic algorithms. The DSA algorithm was used to optimize seismic support locations for a simple piping system while modeling most of the issues that might arise in a real piping system design. Gupta *et al.* [8] have conducted a study on such problems using genetic algorithms and MGA techniques. The sample piping system discussed in their work has been approached using the DSA algorithm.

6.2.1 Piping System Design

Piping systems are designed considering dead weight, thermal stresses, and hydraulic forces. Apart from designing for these loads, supports need to be provided in order to prevent failure in the event of an earthquake. These supports are provided as snubbers, which are special devices that provide support only during a seismic event. These snubbers are associated with high initial cost, high maintenance cost and depending on the industry, extemely high outage costs in the event of failure. Hence optimizing pipe support systems can be of great practical value. The optimization procedures should address the following concerns

- reducing the number of supports as this would lead to reduced initial investment.
- ensuring that stresses developed satisfy design requirements
- ensuring that displacements satisfy the serviceability requirements
- ensuring that the fundamental frequency of vibration of the piping system is more than 1 Hz. In general, piping systems are designed to have a natural frequency of 5 to 10 Hz in order to avoid being in resonance with either the earthquake input or the the frequency of vibration of the equipment, like pumps and motors, mounted on the piping system.
- ensuring that support capacities are in exceedence of the support loads.

Description

Figure 6.8 gives a skeletal diagram of the piping system being considered. The piping system consists of tube of 46 cm (18 inches) outer diameter and a wall thickness of 3.5 cm (1.275 inches) anchored at both ends. The pipe has a number of valves and pumps, motors which are modeled as lumped masses. Table 6.1 lists these lumped masses and their locations. The finite element model used to analyze the piping system consists of 46 nodes with 2 degrees of freedom at each node (vertical displacement and in-plane rotation). The piping system was analyzed using a commercial piping system analysis software called PIPESTRESS [7].

Optimization of 7 piping systems all having the same skeletal structure as shown in Figure 6.8, but each having different number of supports, was performed. In each system, the number of supports is fixed and the optimal location of snubbers was obtained with capital cost as the objective value.

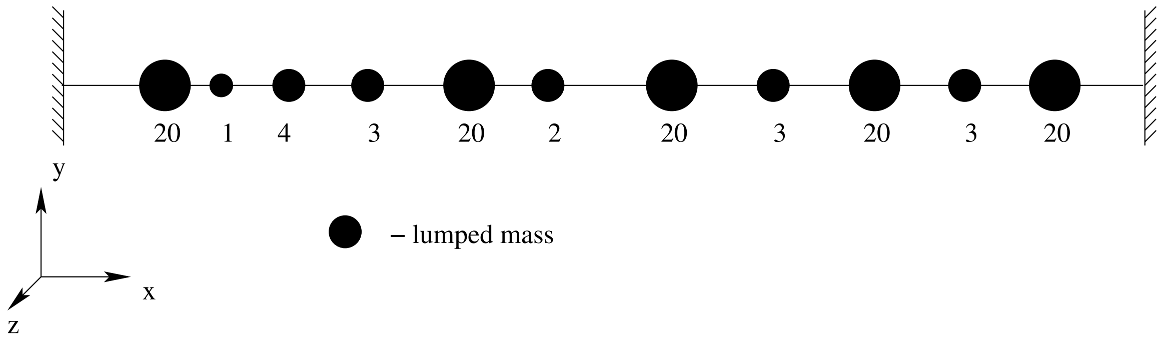

Figure 6.8: Pipe System Skeletal Model

Sl. No	Distance from left end of the pipe (in ft)
1	12
2	18
3	30
4	42
5	54
6	63
7	75
8	87
9	102
10	111
11	129

Table 6.1: Lumped Mass Distribution on Piping System

The solution is represented as an array of binary decision variables, where each variable indicates the existence of a support at a location if its value is 1. During the analysis, rigid supports are assumed at all the support locations and maximum loads at these locations for the earthquake loads were used to choose sufficiently high capacities for the respective supports. Solution transitions are performed by randomly choosing an existing support and moving it to previously vacant support location.

Sl. No	Support Capacity(lbs)	Cost(dollars)
0	0	0
1	1978	15000
2	2067	15100
3	3600	21000
4	8610	21500
5	13750	24000
6	19150	24600
7	20100	24800
8	23450	32000
9	37300	35500
10	62000	36000
11	70350	42000

Table 6.2: Support Capacities and Capital Costs

The objective value was represented as

$$\frac{C + a\Delta\delta + b\Delta\sigma}{d} \tag{6.1}$$

$$\Delta\delta = \sum_{\substack{i=1 \\ s_{i\ max} > s_{all}}}^{n_s} s_{i\ max} - s_{all}; \Delta\sigma = \sum_{\substack{j=1 \\ \sigma_j > \sigma_{all}}}^{n} \sigma_j - \sigma_{all}; C = \sum_{i=1}^{n_{sup}} c_i; \tag{6.2}$$

where a, b are chosen depending on how rigidly the respective constraints are to be enforced, d is used to scale down the high objective values for this problem, σ_j is stress at node j, σ_{all} is allowable stress, $s_{i\ max}$ is maximum displacement to span ration in span i, s_{all} is allowable displacement to span ratio, n_s is number of spans in the piping system, n is the number of nodes, n_{sup} is the number of supports and C_i is the capital cost of support at location i. a, b, d were emperically chosen to be big enough to ensure that only feasible solutions are obtained as final solutions while allowing the algorithm as much freedom as possible to explore the search space. Based on these criteria, a, b, d were emperically determined to be 5000, 3600 and 100 respectively.

The initial temperature for various cases was chosen such that the initial acceptance rate was about 80%. A temperature reduction factor of 0.95 and 80 maximum allowable transitions were used for all the cases. The maximum number of attempted transitions at each temperature ($N_{all\ trans}$) was set at $\frac{2}{3}$ times the neighborhood size of the problem. For example, for a piping system with 3 supports,

$$N_{all\ trans} = \frac{2}{3}.{}^{3}C_1.{}^{43}C_1$$
$$\approx 80$$

As the SA algorithm is a random process, averages of values obtained using 5 different seeds was used for analyzing the performance of the algorithm.

Results

The DSA algorithm was applied to a number of piping systems, each with a different number of supports. Table 6.3 lists the various systems and results obtained as compared to those obtained by Gupta *et al.* [8]. In general, it can be observed that the average captial cost obtained using DSA algorithm is close to or better than those obtained using genetic algorithms.

No. Supports	GA	SA(sequential)	SA(2 slaves)	SA(4 slaves)	SA(8 slaves)
3	108000	108000	108000	108000	108000
4	131600	135160	132080	132080	134200
5	162900	164300	163900	160020	159960
6	189000	186460	183840	182180	182820
7	203300	199880	199880	199560	199560
8	218300	217920	217820	217700	217760
9	232700	234500	232820	232860	232820

Table 6.3: Average Capital Costs

Total enumeration of all possible solutions was performed for the 3, 4 and 5 support cases. The DSA algorithm finds this optimal solution consistently for the 3 support case. Table 6.4 provides a comparison of the best capital cost obtained for various cases using the DSA algorithm.

Table 6.5 tabulates the time taken for executing the algorithm for various cases. It can be seen that the most time taken is about 3 hours (sequential SA for 9 support case) and all the runs with 8 slaves completed execution in less than 25 minutes, as compared

No. Supports	GA	SA(sequential)	SA(2 slaves)	SA(4 slaves)	SA(8 slaves)
3	108000	108000	108000	108000	108000
4	131600	132600	131600	131600	132600
5	162900	164100	163600	156900	156900
6	189000	183100	181200	181200	180500
7	203300	197200	197200	195600	195600
8	218300	217700	217700	217700	217700
9	232700	232700	232700	232700	232700

Table 6.4: Best Capital Costs

to an execution time of about 3 hours when executed on 8 slave processors for the genetic algorithm [14, 13].

No Supports	SA(sequential)	SA(2 slaves)	SA(4 slaves)	SA(8 slaves)
3	3579	2279	1029	441
4	2768	1759	905	428
5	4205	2438	1133	571
6	6281	3701	1556	718
7	7234	4616	2081	1045
8	9870	5955	2796	1242
9	10988	8087	3181	1569

Table 6.5: Time in Seconds for various cases

Figure 6.9 gives a graphical representation of the solution quality for various cases as compared to execution time. Each contour on the graph represents a different piping system and the points on each contour correspont to results obtained using 8, 4, 2 slave processors runs of DSA algorithm and Sequential run of the DSA algorithm respectively.

The variation in average capital cost for any particular piping system when obtained from using various number of servers was between 2% to 3% of the objective value. Thus, it can be seen from Figure 6.9 that for almost the same quality of solution, the distributed algorithm shows a consistent order of reduction in execution time. Figure 6.10 shows the average solution quality along with the range of objective values found for each of the cases. It can be seen that the variation for any given case over various seeds is very small

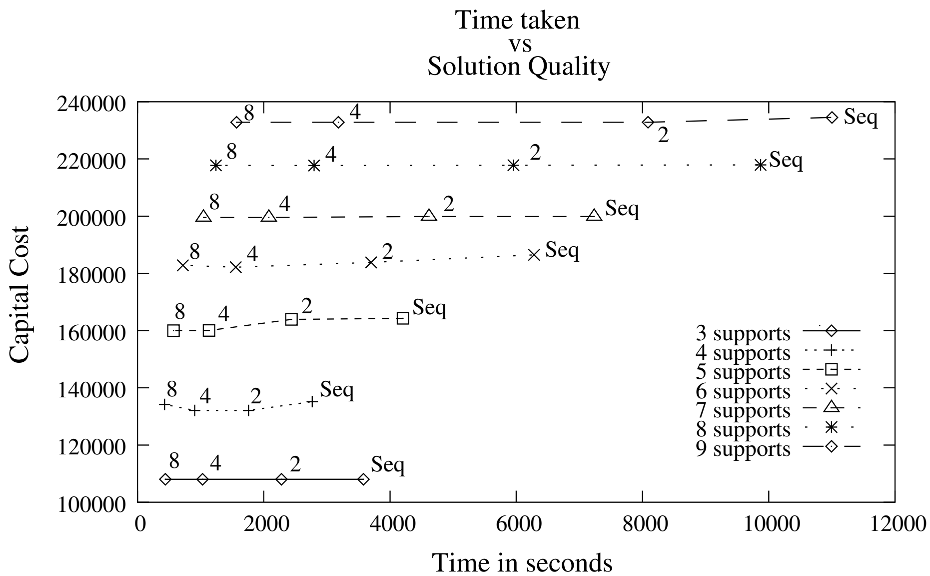

Figure 6.9: Solution Quality vs Time

(maximum of about 5% of capital cost for 4 support case). Thus results are consistent and in similar range for all the cases.

Table 6.6 gives details of the speedup obtained for various cases of the Pipe Support problem.

No Supports	SA(sequential)	SA (2 slaves)	SA (4 slaves)	SA (8 slaves)
3	1	1.57	3.47	8.11
4	1	1.57	3.05	6.45
5	1	1.72	3.71	7.36
6	1	1.69	4.03	8.73
7	1	1.56	3.47	6.92
8	1	1.65	3.52	7.94
9	1	1.35	3.45	7.00

Table 6.6: Speed up for various cases

Figure 6.11 presents a graphical representation of the speedup obtain while using the DSA algorithm for various cases of the piping system design.

Figure 6.11 shows the speedup obtained for various piping systems using different number of servers. It can be seen that in most cases the speedup is almost linear and the

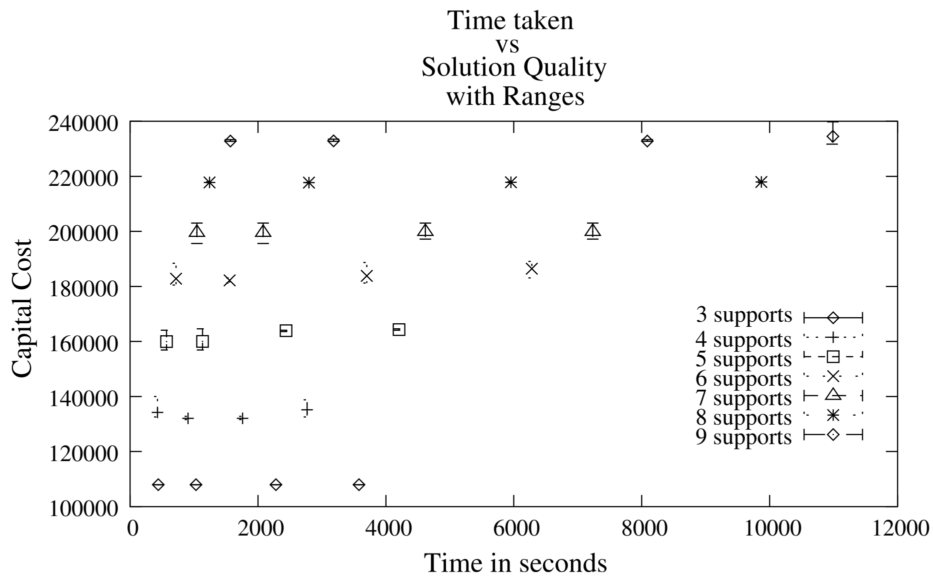

Figure 6.10: Solution Quality vs Time

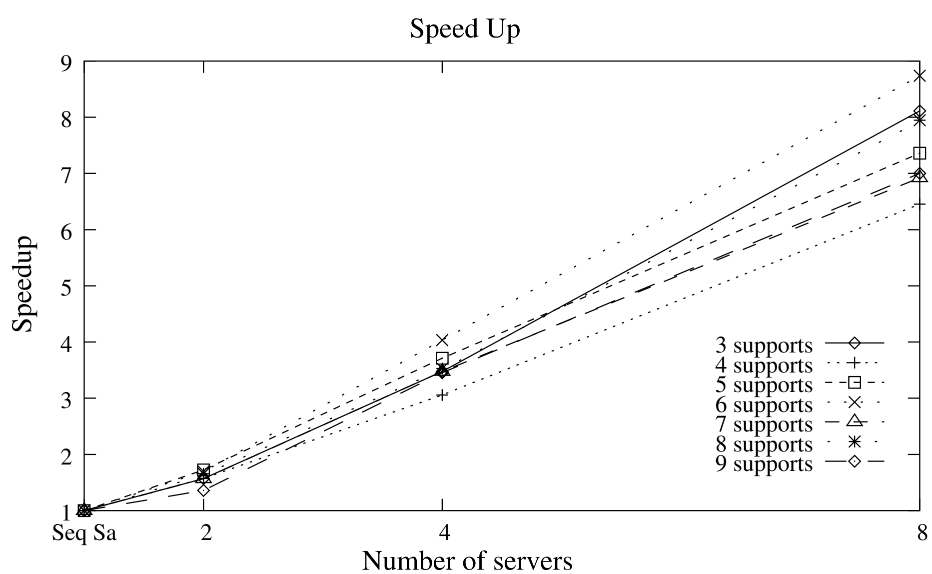

Figure 6.11: Speedup

maximum speedup seen for most piping systems is about 6.5 to 7.8. However, in the case of 3 and 6 support piping systems, we see a super-linear speedup. This behavior can be explained as follows.

Figure 6.12 illustrates the flow of information from the master processor to the slave processors and vice-versa. It can be seen from the Figure 6.12 that frequently, the MCs are halted and the slave processors report current solution to master processor, which is overridden by the current best solution held by the master processor, after each such commuunication. Thus, if a slave processor finds the globally optimal solution, it is reported to the master processor in the next update. When any other slave processor sends an update or requests a solution, it is sent a copy of this optimal solution. Thus once even one of the MCs finds the optimal solution, every subsequent stretch of the MC on every slave processor starts with this optimal solution. Therfore, each of the MCs gets "stuck" in the globally optimal solution. Figure 6.13 shows that the attempted transitions reduces from 2 to 4 to 8 server cases for many piping systems. This means that a smaller amount of work is done in 8 the server case as compared to the 4 server case and so on.

This reduction in the total number of transitions attempted, combined with distributing the computations among more processors, could lead to a super-linear speedup. It can be seen that for all the runs of the 3 support system, the optimal solution is found, which implies that the above described effect does come into play as the global optimum is in fact encountered by at least one of the slave processors in all the cases.

We can see that, as compared to the knapsack problem, the speedups obtained for various cases of the pipe support optimization problems are higher. This could probably be attributed to the fact that the pipe support system problem is more coarse grained than the knapsack problem as it involves calling system processes, performing a considerable amount of file I/O, and executing PIPESTRESS for each function evaluation. Also, the search space in the case of the piping system is much smaller (${}^{46}C_3$ for the 3 support case) as compared to 2^{40} in the case of the knapsack problem.

Figure 6.15 presents a graphical representation of efficiency for various cases while using the DSA algorithm for various piping systems.

Table 6.7 tabulates the results presented in Figure 6.15. It can be seen that the DSA algorithm shows a high efficiency, in all cases above 0.70, consistently. Thus the algorithm is

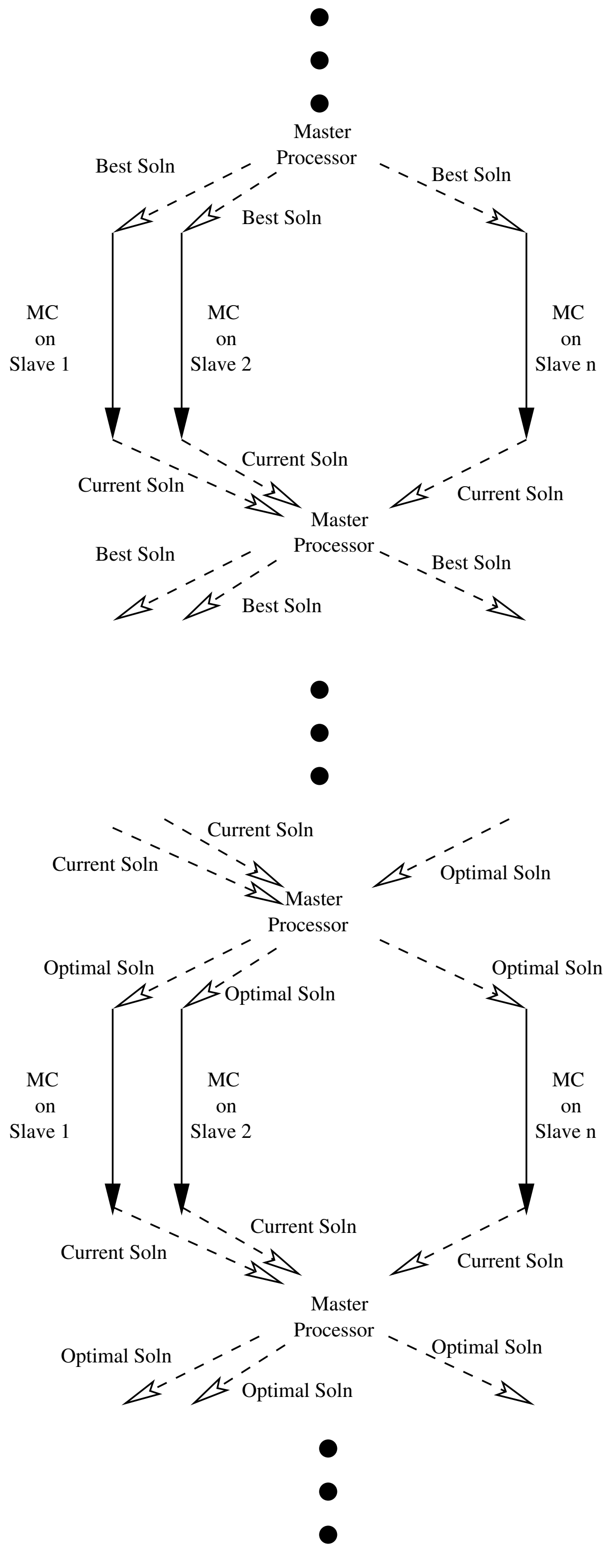

Figure 6.12: Model for information flow between Master - Slave processors for the 3 support case

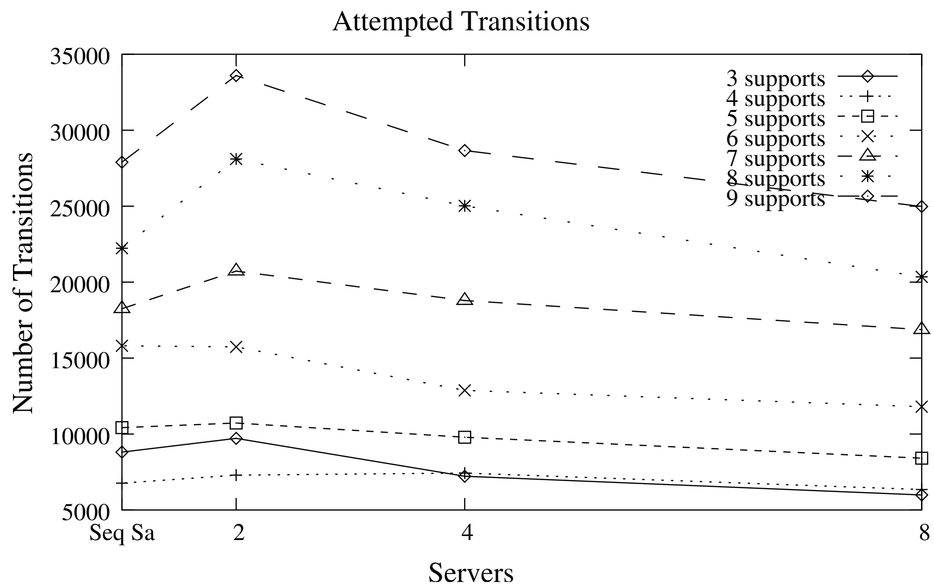

Figure 6.13: Attempted Transitions

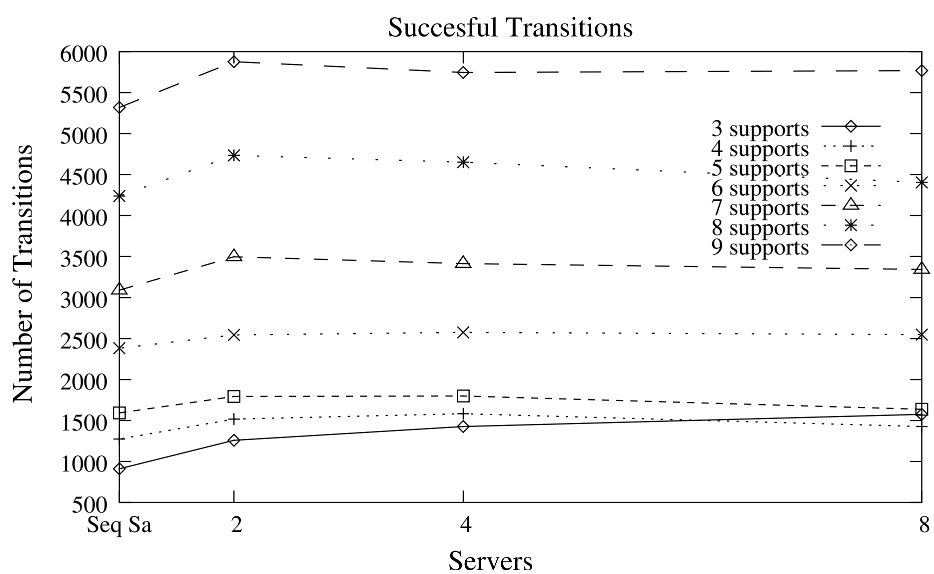

Figure 6.14: Successful Transitions

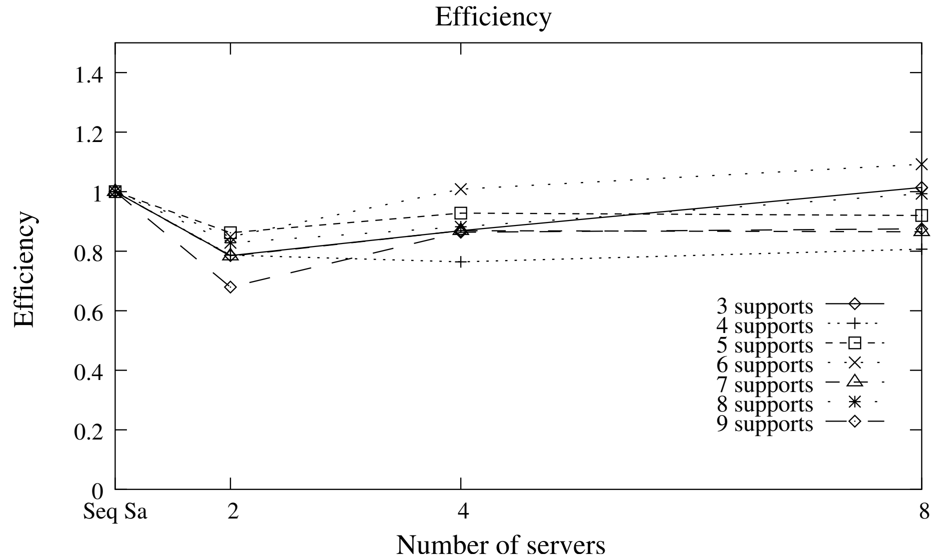

Figure 6.15: Efficiency

scalable and shows a proportionate increase in speedup as the number of servers is increased.

No Supports	SA(sequential)	SA (2 slaves)	SA (4 slaves)	SA (8 slaves)
3	1	0.78	0.86	1.01
4	1	0.78	0.76	0.80
5	1	0.86	0.92	0.92
6	1	0.84	1.00	1.09
7	1	0.78	0.86	0.86
8	1	0.82	0.88	0.99
9	1	0.67	0.86	0.87

Table 6.7: Efficiency for various cases

Chapter 7

Conclusion

It can be seen that the DSA algorithm is effective in solving computationally intensive optimization problems. The distributed algorithm in several cases shows an order of magnitude increase in the quality of the solution. Apart from good solution quality, it can also be seen that the primary aim of speeding up the process of simulated annealing is achieved efficiently. Depending on the problem, various levels of speedup are observed and the algorithm has been found to be reasonably scalable as well. Thus it can be seen that with a judicious choice of parameters, the DSA algorithm can be applied effectively to various problems.

7.1 Future Work

The following are a few suggestions for future work on this framework

- As has been discussed before, the number of attempted transitions on a MC (after step down) is a vital parameter, which has to be determined in a more concrete manner, preferably in terms of the neighborhood size of the problem.
- A number of parameters should be defined more precisely in terms of other deterministic parameters of the problem, such as neighbourhood size or search space. These include, step down factor for attempted transitions at each temperature, update frequency, number of non improving temperature-runs to determine if the MC is stuck, number of non-improving temperature transitions to determine if the MC is frozen, which can

- The update frequency may also be arrived at using the convergence properties of Markov chains.
- The pipe support optimization problem indicates that the behavior of the DSA algorithm can be superlinear in some cases problems. Further study into the effect of search space and computational intensity of the function evaluations to further understand and possibly use this behavior of the DSA algorithm.
- The framework can be made more user friendly by adding a graphical user interface to facilitate the use of the framework and to monitor the progress of the algorithm on various servers.
- The DSA framework seems to show satisfactory scaleability on the homogenous 8 computer cluster. Further investigation can be performed on the scalability of the algorithm over a larger number of servers and on other architectures.

List of References

[1] M. Baker, B. Carpenter, S. Ko, and X. Li. mpiJava: A Java Interface to MPI. In *Proceedings of Europar*, 1998.

[2] V. C. Barbosa and E. Gafni. A Distributed Implementation of Simulated Annealing. *Journal of Parallel and Distributed Computing*, 6:411–434, 1989.

[3] J. Bull and M. Kambites. JOMP – An OpenMP-like Interface for Java. In *Proc. of ACM Java Grande 2000 Conference*, pages 44–53, 2000.

[4] B. Carpenter, Y. J. Chang, G. Fox, D. Leskiw, and X. Li. Experiments with HP Java. *Concurrency: Practice and Experience*, 9(6):633–648, 1997.

[5] T. Chiba, S. Okada, I. Fujii, and K.Itami. Optimum Support Arrangement of piping systems using Genetic Algorithms. *Journal of Pressure Vessel Technology*, 118:507–512, November 1996.

[6] M. D. Cunha and J. Sousa. Water Distribution Network Design Optimization: Simulated Annealing Approach. *Journal of Water Resources Planning And Management, ASCE*, 125(4):215, 1999.

[7] DST Computer Service, Geneva, Switzerland. *DST/PIPESTRESS User's Manual*, version 3.4 edition. 1992.

[8] A. Gupta, P. Kripakaran, G. Mahinthakumar, and J. W. Baugh Jr. GA-Based Decision Support for Optimizing Seismic Response of Piping Systems. *ASCE Journal of Structural Engineering*, accepted 2003.

[9] O. Hasancebi and F. Erbatur. Layout Optimization of Trusses using Simulated Annealing. *Advances in Engineering Software*, 33:681–696, 2002.

[10] J. W. Baugh Jr., G. R. Kakivaya, and J. R. Stone. Intractability of the Dial-A-Ride Problem and A Multiobjective Solution using Simulated Annealing. *Engineering Optimization*, 30:91–123, 1998.

[11] J. W. Baugh Jr. and S. V. Kumar. Asynchronous Genetic Algorithms for Heterogeneous Networks using Coarse-grained Dataflow. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2003)*, 2723:730–741.

[12] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, May 1983.

[13] P. Kripakaran. GA-Based Decision Support for Optimizing the Response of Secondary Systems. Master's thesis, North Carolina State University, 2002.

[14] P. Kripakaran. Personal Communication, May 2003.

[15] S. Y. Lee and K. G. Lee. Synchronous and Asynchronous Parallel Simulated Annealing with Multiple Markov Chains. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):993–1008, 1996.

[16] J. P. B. Leite and B. H. V. Topping. Parallel Simulated Annealing for Structural Optimization. *Computers and Structures*, 73:545–564, 1999.

[17] J. A. Mathew, H. A. James, and K. A. Hawick. Development Routes for Message Passing Parallelism in Java. In *Proc. of ACM Java Grande 2000 Conference*, pages 54–61, 2000.

[18] N. Mohamed, J. Al-Jaroodi, H. Jiang, and D. Swanson. JOPI: A Java Object Passing Interface. In *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pages 37–45, 2002.

[19] T. M. Nabhan and A. Y. Zomaya. A Parallel Simulated Annealing algorithm with Low Communication Overhead. *IEEE Transactions on Parallel and Distributed Computing*, 6(12):1226–1233, December 1995.

[20] D. Janaki Ram, T. H. Sreenivas, and K. Ganapathy Subrahmaniam. Parallel Simulated Annealing algorithms. *Journal of Parallel and Distributed Computing*, 37:207–212, 1996.

[21] P. Roussel-Ragot and G. Dreyfus. A Problem Independent Parallel Implementation of Simulated Annealing: Model and Experiments. *IEEE Transactions on Computer-Aided Design*, 9(8):827–835, 1990.

[22] E. E. Witte, R. D. Chamberlain, and M. A. Franklin. Parallel Simulated Annealing using Speculative Computation. *IEEE Transactions on Parallel and Distributed Systems*, 2:83–494, October 1991.

Appendix A

Using DSAFramework

Using the DSA framework to solve an optimization problem involves 3 steps

- Extending the `Matter` class to implement the solution representation
- Editing the `DistSa_info.txt` file to choose the parameters for the DSA Algorithm
- Executing the program.

A.1 Extending the `Matter` Class

In order to utilize the DSA framework to solve an optimization problem, users must implement a class which extends the matter class and implement methods to perform various simulated annealing operations for their particular problems. The following is skeletal code of one such user-defined class.

```
public class UserClass extends Matter {

  //define a solution representation, variables
    ...

  //define constructors
    ...

  //implement method to return the initial temperature
  public double getInitTemperature(){
    ...
  }
```

```
  //implement method to randomize the solution representation
  public void randomize(){
    ...
  }

  //implement method to perform solution transitions
  public void doTransition(){
    ...
  }

  //implement method to undo the transitions made
  public void undoTransition(){
    ...
  }

  //implement method to return the Energy of the Matter
  public double analyze() {
    ...
  }

  //implement the clone method
  public Object clone(){
    ...
  }
}
```

A.2 Input File

The two main classes of the DSA framework (`DistributedSA`, `SimulatedAnnealing`) read the file DistSa_info.txt before being initialized to obtain the values of various parameters that govern their behavior. The user can modify the contents of this file to customize the behavior of the framework.

When the classes read the DistSa_info.txt file, the lines beginning with a # mark are ignored. The parameters can be specfied in any order. The format for defining any parameter is to specify the name of the parameter, followed by at least one white space, followed by a colon(:), followed by another white space, followed by the value. The following is a sample input file used specify parameters for solving a knapsack problem using `KnapsackProblem` class as the `Matter`instance.

```
### Input file for running the Distributed Simulated Annealing

### Input parameters for the DistributedSA class

# No. of markov chains
no of chains                    : 1

# The seed for this run -1 => unseeded run
Seed                            : 1000

### Input/control parameters for the SimulatedAnnealing class

# Name of matter class to use
Matter class                    : KnapsackProblem

# If synchronised SA to be used
Synchronised                    : True

# Factor for reducing the max_attempted transitions
# in the case of distributed sa
# use a value of -1 if you do not want to step down
Step factor                     : 1

# Max attempted transitions at each Temperature
Max attempted transitions       : 20

# Intial temperature for the annealing scheme
# -1 => temperature will be calculated
Intial temperature              : 80

# Temperature reduction factor for the annealing scheme
Temperature reduction factor    : 0.90

# Number of succesive Temperature transitions after which
# to send an update to the Client.
Update Frequency                : 10

# Number of succesive non-improving Temperature runs
# after which the markov chain can request an update
Stuck Limit                     : 3

# Number of succesive non-improving Temperature runs
# after which the markov chain is deemed frozen
Frozen Limit                    : 5
```

A.3 Executing the Program

Executing the DSA framework involves starting the `DistributedSA` program on a computer meant to act as the master processor and starting the `Server` program on machines meant to act as the slave processors.

The syntax for executing the `DistributedSA` program is as follows.

```
java DistributedSA <portnumber> <distributed>
```

where the command line argument

- `<portnumber>` is the number of the port on which pool of tasks should communicate with the slave processors. This number can be any number greater than 1024 and less than 65535. [1]
- `<distributed>` is a boolean flag which indicates whether the DSAFramework should operate in a distributed fashion or not. If this argument provided is `false` the DSA framework executes a Sequential Simulated Annealing algorithm to solve the problem. If this argument provided is `true`, the DSA framework executed the Distributed Simulated Annealing algorithm.

The syntax for executing the `Server` program is as follows

```
java Server <client> [port]
```

where the argument

- `<client>` specifies the IP address of the computer which acts as the master Processor.
- `[port]` specifies the port number on the master processor on which to communicate with DSA framework. This should be identical to the number used for the `port` argument used for starting the `DistributedSA` program.

[1] Since there are a number of services running on various ports, it would be advisable to choose a TCP/IP port number not registered for any other purposes. The url `http://www.iana.org/assignments/port-numbers` provides detailed information about the TCP/IP ports which have been registered for network services.