Task Assignment for Workstation Farms

Xiaohong Yang

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfilment of the Requirements for

the Degree of Master of Computer Science at

Concordia University

Montréal, Québec, Canada

January 1994

ISBN 0-315-90888-2

ABSTRACT

Task Assignment for Workstation Farms

Xiaohong Yang

Due to the existence of large quantity of interconnected workstations (workstation farm) and their low utilization, it is very attractive to use the idle CPU cycles of these workstations to simulate a parallel computer. The performance of such simulation depends heavily on the assignment of the interacting task modules of a parallel program to the processors. A good assignment should balance the work load of the processors while minimizing the interprocessor communication overhead on the network, which is usually the bottlekneck of the system performance. In this thesis we study the task assignment problem for workstation farms with various configurations and design efficient heuristics to produce assignments to minimize the completion time of parallel programs.

We first formulate the task assignment problem for five different configurations of workstation farms. We study the mechanisms of simulated annealing and tabu search and propose a new general technique called *stochastic probe* for combinatorial optimization which combines the advantages of both the stochastic search in simulated annealing and the aggressive search in tabu search. Heuristics based on these three techniques are proposed for the task assignment problem based on different optimization models. Extensive experiments demonstrate that our stochastic probe heuristics are superior to the other techniques in both solution quality and CPU time.

To further reduce the computation time, we study the parallelization of the above three techniques. We parallelize our sequential task assignment heuristics and run them on butterfly GP-1000. Experiments show that for up to 10 pocessors our parallel stochastic probe heuristics achieved almost linear speedup with solution quality comparable to those of their corresponding sequential versions.

Acknowledgments

I would like to express sincere gratitude and appreciation to my thesis supervisor Dr. Lixin Tao, for his extensive guidance and patience throughout my graduate studies, and supporting me financially to complete my degree.

I would also like to thank Prof. Yongchang Zhao, for giving me every help during my research work for the thesis.

Special thanks to my husband, for the sacrifices, moral support and the patience to put up with my every whim during all this time.

More than anyone else, I would like to thank my parents. It was their continued support and encouragement that made this degree possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Workstations have been extensively used in most institutions and companies. But most of them are active less than 10 percent of time [43]. With the advent of high-speed optical cables, many researchers and companies are proposing to use the new cables to connect the existing personal workstations to form a workstation farm, which can:

- provide facilities for utilizing remote computer resources or data not existing in local computer systems;
- increase the throughput by providing the facilities for parallel processing;
- provide users with the modality, flexibility, and reliability.

But the potential of the enhanced system performance depends on several factors:

- **Hardware**, such as the speed of the processors, memories, and interconnection network;
- **Application program**, the amount of parallelism in the application program;
- **Program mapping**, include task partition and task allocation. The task partition and task allocation activities influence the load of interprocessor communication and potential for parallelism.

Many practical and theoretical problems must be solved to realize the potential of the workstation farms [39].

One serious problem is the degradation in throughput caused by the saturation effect [13]. Ideally, we would expect throughput increases linearly as the number of processors increases in a multiple processor environment. For example, if there are n processors, and each has the processing speed of k, then we would expect the system processing speed to be nk. But in practice, the throughput in a multiple processor system increases significantly only for the first additional processors. At some point, the throughput actually begins to decrease with each additional processor. This is called "saturation effect", which is caused by many factors, such as control overheads, excessive interprocessor communication, unbalanced load, queueing delay and the precedence order of the parts of a task assigned to separate processors, among which the excessive interprocessor communication and unbalanced load are two key factors that affect the low processing speed of the system.

The saturation effect will still exist even in the distributed system connected by high-speed optical cables. Because, even though optical cables have reduced the overhead incurred by the inter-processor communications, the message transmission speed on these cables is still lower than that of the ever-improving processor technologies (including optical computing). Since the bus can transmit only one message at a time, the sequential message passing can sequentialize the computation and constitute a bottle-neck of the global system performance.

In order to avoid the saturation, we must eliminate or minimize these inhibiting factors. Task partitioning and task allocation are two steps necessary to eliminate or minimize these factors, especially the excessive interprocessor communication. *Task partitioning* can be considered as a software design issue, which refers to the break-down of a processing task into several individual modules with minimum intermodule communication. *Task allocation* is the process of allocating modules to processors in such a way that the interprocessor communication is minimized while the computation load is balanced in the system. In this thesis, we will focus on the task allocation problem. We assume that the task partitioning process has been performed by the software designer and that each task that arrives in the distributed system is already partitioned into a set of modules. For convenience and consistency, in the rest of the thesis, we will use term "task assignment" instead of "task allocation".

Task allocation has two classes of policies: static and dynamic. It depends on the time at which the scheduling or allocation is carried out. *Static task allocation* is a priori allocation of tasks to the processors; it depends on the average behavior of the system and not its current one, and the allocation does not change during the life-time of the tasks. *Dynamic task allocation* is run-time scheduling; the processors exchange load information periodically and migrate tasks on a dynamic basis. Since last decade, many researchers have worked on the task allocation problem, some on static, some on dynamic. Each has its own considerations.

However, the dynamic allocation problem is much more sophisticated than the static one, and its overhead is more significant. On the other hand, many of the parallel algorithms which we have been working on have static properties, such as scientific problems, engineering problems, etc.. Therefore, in this thesis we focus on the static task assignment for the workstation farms, in which workstations can be homogeneous or heterogeneous, and the links connecting each pair of workstations can also be homogeneous or heterogeneous. In this thesis, five models are studied for different configurations of workstation farms.

Since our studies on the task assignment problem are to improve system's performance, mainly the speed, the execution time for solving the task assignment problem itself should be as short as possible. In this thesis, we first emphasize on finding good sequential solution searching heuristics (algorithms), which mainly depend on good searching strategies, neighborhood design, and move set design. Then, based on the good sequential heuristics which we found, we try to find a good strategy to parallelize it to take advantage of the existing parallel computing facilities to further reduce the execution time of the task assignment heuristics.

This thesis is organized as follows:

Chapter two first gives the general formalization of the task assignment problem for our workstation farms, then reviews some models encountered in the literature. The related solution techniques for these models in the literature will be discussed briefly. After analyzing these models, we present five models for different kinds of workstation farms: 1) uniform m-way graph partition model, 2) nonuniform single-bus total cost model, which uses total cost as objective function, 3) nonuniform single-bus completion cost model, which uses completion cost as objective function, 4) general nonuniform total cost model, which uses total cost as objective function, 5) general nonuniform completion cost model, which uses completion cost as objective function.

Chapter three mainly discusses some solution techniques. After analyzing the existing techniques in the literature, we emphasize on three heuristics: 1) simulated annealing, 2) tabu search and 3) our stochastic probe search. We first study the two general combinatorial methodologies: simulated annealing and tabu search, and then present our adaptation of these two techniquess to our task assignment problem. Since simulated annealing is too randomized while tabu search is too aggressive, we develop our stochastic probe approach which combines the advantages of these two heuristics. To further reduce the execution time for solving the task assignment problem, we study some strategies for parallelizing each of the three heuristics. We provide parallelization of simulated annealing, tabu search, and our stochastic probe approaches in this chapter. In the last part of this chapter, we address our experimental environment, our benchmark graphs involved in the corresponding experiments, and our parameters for each approach.

Chapters four, five, six, seven and eight address the adaptation of the above three heuristics to the task assignment problem on the five models we designed for our workstation farms in chapter two respectively. Our experimental analysis, comparison of their performances, and observations are also presented in each of these chapters for the corresponding models.

Chapter nine comes to the conclusion of this thesis work.

Chapter 2

Task Assignment Models

Since last decade, many researchers have been working on the task assignment problem. Different models for different targeted systems have been proposed, varying from simple ones to complicated ones. In this chapter, we will first review the models encountered in the literature, and then present our models for our workstation farms.

2.1 Model Review

Consider a set of N processes which is to be allocated to a set of M processors. The execution cost $e_{i,j}$, which is given by an $N \times M$ execution cost matrix, represents the cost of executing process i on processor j, while the communication cost $c_{i,j}$ between processor i and processor j is given by an $N \times N$ communication cost matrix. An objective function is usually defined in terms of these two kinds of costs. A good task assignment should minimize this objective function.

The above model can also be enhanced by including constraints which make it more realistic. Such constraints refer to memory and occupancy, precedence relationship of modules in a program, system response time, replication etc..

The models encountered in the literature vary from simple ones, where the sole aim of the model is to assign a given number of processes to 2 processors, to more complex ones, where one tries simultaneously to meet constraints on the availability of processor resources. These models can be classified into three categories: 1) graph theoretic, 2) 0-1 integer programming, and 3) queueing theory. In this section, we review different kinds of models by going through these three categories classified above. All the algorithms proposed with their corresponding models in the literature will be discussed briefly along with the models.

2.1.1 Graph Theoretic Models

Most of the Graph Theoretic models were proposed for loosely-coupled distributed systems, in which the inter-processor communication cost is significant, while the the communication within a processor is usually negligible. Within this graph theoretic category, several kinds of models are presented in the literature: 1) network flow model, 2) graph matching model, 3) the layered, doubly weighted graph model, and 4) graph partition model.

Network flow model

Network flow model was proposed by Stone [40, 39] to solve task allocation problem for dual-processor systems. In the network flow model, Stone assigned tasks to two processors and used undirected graphs to depict execution and communication. The modules are represented by a set of nodes in the graph and the inter-module communication costs are represented by the weights on the edges of the graph. Two kinds of costs are assumed in the model: execution costs and communication costs. The objective function is the sum of all the active costs for an assignment. In other words, the task assignment strategy is to minimize the total cost. No resource constraints are imposed on any of the processors. In this model, each processor is assumed to have infinite memory to store all the modules assigned to it and their data. In general, any process is free to reside on any processor.

In order to use the max-flow/min-cut algorithm, the graph needs to be first transferred to commodity flow graph by adding two more nodes P_1 and P_2 to the graph, which correspond to the two processors, to represent the execution costs with respect to each processor. The weight on the edge connecting a module node to P_2 is the execution cost of the corresponding module on processor P_1, and the weight on the edge connecting P_1 to a processing node is the execution cost of that module on processor P_2. Then, the max-flow/min-cut algorithm is performed on the commodity flow graph so that the modules are optimally partitioned into two sets, each allocated to a particular processor. The value of the min-cut represents the minimum amount of the inter-processor communication.

Although this method is attractive in its simplicity, it has several serious limitations.

- It is for two processor system only. In general, an extension of this method to an arbitrary number of processors requires an n-dimensional max-flow/min-cut algorithm which quickly becomes computationally unattractive. This will limit the usefulness of this method in many applications.
- The model does not encourage concurrency. All the task models may be assigned to the same processor.
- The system considered in this method has no system resources constraints at all. It provides neither a mechanism for representing limited resources in memory size or processing time, nor a mechanism for load balance, nor mechanism for the preservation of data flow precedence.
- The maximum flow algorithm can only treat edges of the graph with nonnegative capacity. Although Stone made some effort to overcome this difficulty, there are still some cases which this method can not deal with [41].

Because of these advantages and disadvantages of this model and method, many researchers continue to work on it, propose several modified models based on this network flow model.

Lo did some work to extend Stone's network flow model to n-processor system [29]. Lo modeled a system of k tasks and n processors as a network in which each processor is a distinguished node and each task is an ordinary node. An edge is drawn between each pair of tasks t_i and t_j and is given the weight c_{ij}, the communication costs between the two tasks. There is an edge from each task node t_i to each processor node p_q with the weight

$$w_{iq} = \frac{1}{n-1}\sum_{r \neq q} x_{ir} - \frac{n-2}{n-1}x_{iq}$$

An n-way cut in such a network is defined to be a set of edges which partitions the nodes of the network into n disjoint subsets with exactly one processor node in each subset and thus corresponds naturally to an assignment of tasks to processors. The cost of an n-way cut is defined to be the sum of the weights on the edges in the cut, which is exactly equal to the total sum of execution and communication costs incurred by the corresponding assignment.

For this model, Lo developed a heuristic algorithm referred to as algorithm A which combines recursive invocation of Max-Flow/Min-Cut algorithms with a greedy algorithm to find suboptimal assignments of tasks to processors. Algorithm A consists of three parts: Grab, Lump, and Greedy. The first part of algorithm A, Grab, produces a possibly partial assignment of tasks to processors by having each processor "grab" those tasks that are strongly attracted to it. If the assignment is complete, it is optimal. However, if some tasks still remain unassigned, Lump, the second part of the algorithm A, tries to find a quick and dirty assignment by assigning all remaining tasks to one processor if it can be done "cheaply enough". If Lump still cannot complete the assignment, Greedy, the last part of the algorithm, is invoked. Greedy identifies clusters of tasks between which communication costs are "large". Greedy merges such clusters of tasks and assigns all tasks in the same cluster to the cheapest processor for that cluster. The resultant assignment may be suboptimal. Good performance of the heuristic is reported in Lo's simulation results.

But the model still uses the total execution and communication costs as the performance criteria to be optimized. Therefore, it keeps the major flaw that no explicit effort is made on concurrency, yielding assignments to utilize only a few of the available processors. For this reason, Lo augmented Stone's model with *interference costs* which are incurred when two tasks are assigned to the same processor. Interference costs reflect the penalty for two tasks to compete for the same resources of the processor assigned. The augmented model consists of three kinds of costs: 1) execution costs, 2) communication costs, as defined before, and 3) the interference costs. An optimal assignment for this model is one which minimizes the total of sum of execution, communication, and interference costs. A homogeneous n-processor system can be modeled as a network in which an n-way cut corresponds to an assignment of tasks to processors. Let the edge from each task node t_i to each processor node p_q have the weight

$$w_{iq} = \frac{i}{n-1}\sum_{r \neq q} x_{ir} - \frac{n-2}{n-1}x_{iq} + \frac{1}{2(n-1)}\sum_{1 \leq l \leq k} I_{il}$$

Let the edge between two task nodes t_i and t_j have the weight

$$c'_{ij} = c_{ij} - I_{ij}$$

For this model, if $I_{ij} < c_{ij}$, $1 \leq i,j \leq k$, the max-flow/min-cut algorithm can be applied to find optimal assignment for two-processor system. For n-processor systems, algorithm A can be applied to find suboptimal assignments. Lo's simulation results show that heuristics designed to minimize total execution, communication, and interference costs yield assignments with a high degree of concurrency. But for arbitrary I_{ij}, Max-Flow/Min-Cut algorithm cannot be invoked. Algorithm A is not a useful heuristic for the model with the interference costs.

To further introduce the degree of parallelism attained by assignments, Lo presented other two models in which optimal and suboptimal solutions are measured by completion costs instead of total costs. Completion costs here refer to as the natural extension to latest finishing time. In the model without interference costs, the completion time is defined as

$$w_f = max_{1\leq q\leq n}\Big(\sum_{f(t_i)=p_q} e_{iq} + \sum_{\substack{f(t_i)=p_q \\ f(t_j)\neq p_q}} c_{ij} \Big)$$

i.e., by Stone and Bokhari [40], the total execution and communication costs incurred on the processor for which these costs are maximal over all processors. Similarly, in the model with the interference costs, completion time is defined as

$$w_f = max_{1\leq q\leq n}\Big(\sum_{f(t_i)=p_q} e_{iq} + \sum_{\substack{f(t_i)=p_q \\ f(t_j)\neq p_q}} c_{ij} + \sum_{\substack{f(t_i)=p_q \\ f(t_j)\neq p_q}} I_{ij} \Big)$$

i.e., the total sum of execution, communication, and interference costs incurred on the processor for which this total is maximal over all the processors.

No algorithm is proposed for either of these two models in Lo's work.

Lee, Lee, and Kim [27] extended Stone's approach to the case for a linear array of any number of processors. Differing from all systems above, the linear array systems considered are not fully connected. The allocation strategy is to minimize the sum of all the active execution and communication costs. Lee and his colleagues solved the task allocation problem for a linear array network by first transferring it into the two-terminal network flow problem, then using the Goldberg-Tarjan's network flow algorithm [27] to achieve the optimal solution.

All the work mentioned above is about static task allocation based on the network flow model. Dynamic task allocation problem based on the network flow model is also studied in the literature.

Based on the network flow model, Stone and Bokhari [40] presented a modified model for dynamically assigning tasks to two-processor systems, where the notion of "phase" is introduced. More information is included in the model such as relocation costs for each module at the end of each phase, costs of residence of the remaining modules for each processor etc. These information is also represented by an undirected graph. The number of nodes in this graph equals to the number of program modules multiplied by the number of phases. The nodes are arranged in a grid with the vertical "columns" of nodes representing the modules and the horizontal "rows" representing the phases. Each individual node represents the residence of a module during a specific phase. Each node is labeled with an upper-case letter which identifies the module it represents and an integer which identifies the phase. The single module that executes during each phase is marked with an asterisk. The "vertical" edges connect successive residences of the same module and the weights of these edge represent the costs of relocating the modules. The "horizontal" edges connect the executing modules with other modules during the same phase and represent intermodule communication costs between the executing module and the other modules. (see Figure 2.1)

As in previous model of Stone, in order to use the Max-Flow/Min-Cut algorithm, this graph needs to be transferred to a dynamic assignment graph by adding two nodes, which represent the two processors. Edges representing the run costs are drawn from P_1 and P_2 to each node representing the executing modules.

In [5], Bokhari shows that a network flow algorithm may be performed in this dynamic assignment graph to get the min-cut, which gives the optimal dynamic assignment of modules, i.e., it specifies which modules are to reside on which processor during each phase.

To extend the dynamic assignment model to n-processor distributed systems, Stone and Bokhari proposed a directed tree model which is only applied to the case in which the intermodule communication pattern is constrained to be a tree. Basically, it is an extension to the above dynamic assignment model when the model is applied to a tree-like structure program. Bokhari designed a dynamic programming approach called shortest tree algorithm for this model [5] to find optimal assignment. However, for this model, there exists a serious problem which is very hard to construct a tree.

All systems considered above are similar to Stone's. They do not have any system

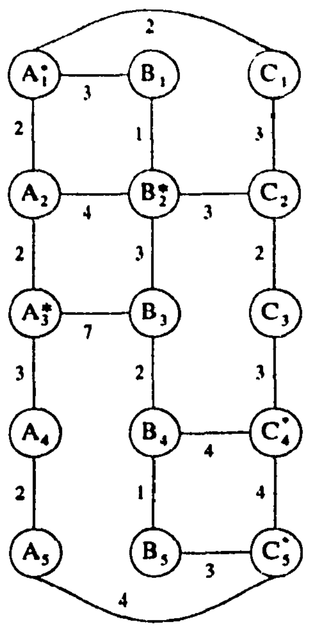

Figure 2.1: Incomplete dynamic assignment graph. The horizontal edges represent communication costs, and the vertical edges represent relocation costs. Asterisk denotes executing module.

resources constraints, either for static or dynamic task allocation model. But in reality, any system's resources are limited, the effect of limited memory size in each processor is usually needed to be considered. Rao and Stone [35] introduced the problem of how to assign modules to a two-processor system with one processor having limited memory capacity so as to minimize the total of execution costs and interprocessor communication costs. It is shown that the processes allocated to the processor with limited memory capacity form a subset of the processes allocated to this processor by the maxflow method used by Stone with no memory capacity constraints. The general problem is NP-complete. To simplify the original network, some reduction techniques by means of condensing certain processes into single nodes, such as techniques based on the Gomory-Hu tree from network flow theory and techniques based on the inclusive cut graph, are suggested. The solution to the original task allocation problem can be found by enumerating the cuts of the inclusive cut graph, which can be very efficient for some problems. But no algorithm of guaranteed polynomial efficiency in the general case is proposed.

Graph Matching Model

Shen and Tsai proposed graph matching model which is based on a graph match approach called weak homomorphism and a cost function representing the maximum time for a task to complete module execution and communication in all the processors [37] . The distributed systems considered here are heterogeneous, either in processors or in communication links, and the processors in the system need not to be fully connected, and there exists little or no precedence relationship or synchronization requirement among the program modules.

In this model, the module relationship of a given task and the processor structure of the distributed system are represented by two undirected graphs, task graph and processor graph respectively. In task graph, each node denotes a module of task, and edge denotes the intermodule communication between the two modules at the ends of the edge. Similarly, in processor graph, each node represents a processor in the distributed system, and each edge represents a communication link between processors. A self-looping edge is added to each node due to the fact that two related modules may be assigned to a single processor. Therefore, the module assignment to system processors is transferred into weak homomorphism graph matching. The optimal task assignment can be found by searching the optimal weak homomorphism which is formalized as a state-space problem. Algorithm A* is applied in search of optimal weak homomorphism. It uses the task turnaround time as the cost measure and minimax criterion of minimization of the interprocessor communication and balance of processor loading as the criteria for the assignment optimization.

However, the cost measure h in Shen and Tsai's algorithm A^* does not give a satisfied lower bound of h^*, which brings the limitation of their scheme that the tree size becomes unwieldy for large problems. The algorithm often requires a large number of evaluations of a complex heuristic function.

A new hybrid strategy is proposed in [34] which combines Stone's maximum flow algorithm and Shen and Tsai's A* algorithm. It uses maximum flow algorithm to compute the cost function for nodes generated during the tree expansion process in order to reduce the number of nodes. This will result in a decreased total runtime even if the maxflow computation takes more time per node. To further cut down the number of nodes in the search tree, dependency among the tasks and consistency among the tentative assignments are taken into account.

Some other heuristics are proposed in the literature to overcome the widely expansion of the tree at each layer in Shen and Tasi's A* algorithm and the time consuming evaluation of the heuristic function [33, 10].

A similar model is proposed by Bollinger and Midkiff [7] for general homogeneous system. A simulated annealing heuristic is presented, with the objective function to minimize the total communication overhead. The objective function is

$$F = \sum_{j,k} c_{jk} + w \max_{j,k} c_{jk}$$

where weight factor w penalizes any configuration that increases $\max_{j,k}(c_{jk})$, varying with temperature. The first term, the total communication, produces a greater variation in cost, making it more desirable as a cost matrix for simulated annealing. While the second term, describing the largest communication, more accurately characterizes the quantity being optimized. Both of total communication cost and largest communication cost are considered in the objective function with factor w. It is reported that the approach can anneal into optimal solution for $N < 128$, where N is the processor number.

The Layered, Doubly Weighted Graph Model

For a single-host, multiple-satellite system, Bokhari proposed the layered, doubly weighted graph model [6]. In this model, all information about execution and computation costs of the modules is included in a layered graph (see Figure 2.2). Each layer corresponds to a processor and the label on each node corresponds to a subchain of modules. Two weights are associated with each edge: a sum weight and a bottleneck weight. In layer k, each edge emanating downward from node $< i, j >$ is first weighted with the time required for processor k to process node i through j. This accounts for the computation time. To the weight on the edge joining node $< a, b >$ in layer k to node $< b + 1, d >$ in layer $k + 1$ is added the time to communicate between modules b and $b + 1$ over the link connecting processors k and $k + 1$. The influence of both the amount of data transmitted between modules b and $b + 1$ as well as the speed of the link between processors k and $k + 1$ can be included in the graph. To take memory constraints on individual processors into account, it suffices to add up the memory requirements of all modules in every subchain. If the sum of memory requirements for nodes i through j exceeds the capacity of processor k, node $< i, j >$ in layer k is deleted, along with all edges incident on it. Any path connecting distinguish nodes s to t corresponds to an assignment of modules to processors. The objective of task allocation in the system represented in the graph is to find the minimum bottleneck path in the graph, that is, of all paths, the one in which the heaviest edge has minimum weight. For this model, Bokhari designed an algorithm named *sum-bottleneck* algorithm which combines the shortest tree algorithm and bottle neck algorithm.

However, this approach can only work under the contiguity constraint that each processor has a continuous subchains of program modules assigned to it. In particular, it can only be applied to chain-structured programs, multiple arbitrarily structured serial programs, and single-tree structured programs, and pipelined programs.

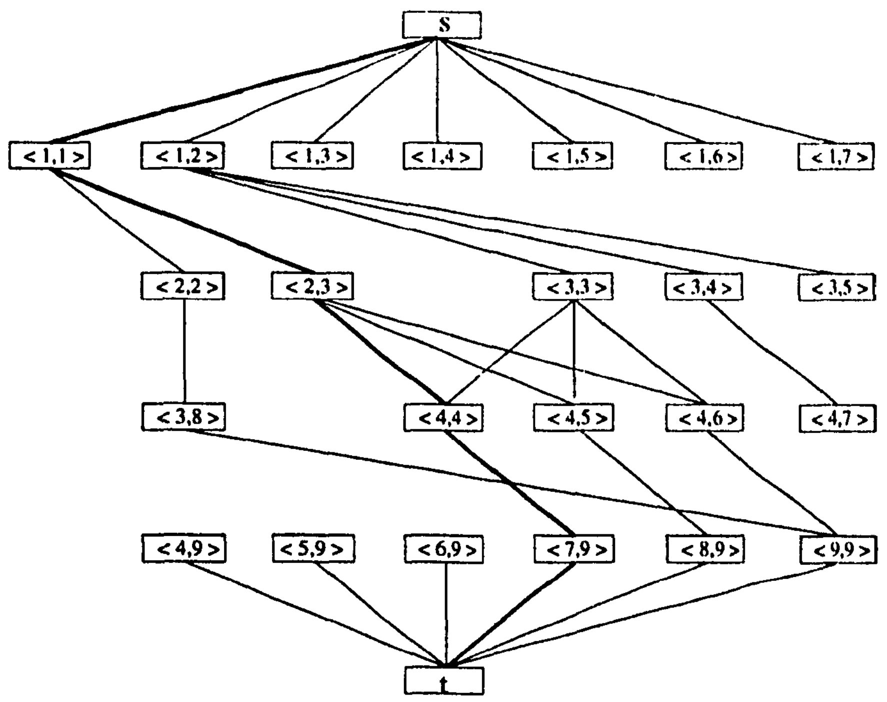

Figure 2.2: The layered graph for a problem with nine modules and four processors.

The other main drawback of the approach is that it is very difficult to construct an assignment tree needed in the sum-bottleneck algorithm.

Graph Partition Model

Graph partition model is a basic graph theoretic model, in which each module of program is modeled as a node in the graph and the communication between two modules is modeled as an edge connecting two nodes corresponding to these two modules. The weights of nodes stand for the execution costs, and the weights of edges stand for communication costs. Researchers have worked on this model for different distributed systems, with different objective functions.

Lo [28] applied this model to an Ethernet-based distributed system, in which processors are identical and there is only one communication pathway. An optimal assignment of tasks to processors in the system is defined by Lo as one which minimizes the total interprocessor communication costs incurred under the constraint of a bound on the maximum number of tasks on each processor. Two algorithms are presented by Lo: *algorithm M* and *algorithm H*. Algorithm M is to find optimal assignments in polynomial time by finding a maximum weight matching in graph, when the number of tasks is less than or equal to twice the number of processors and when each processor may be assigned at most two tasks. Algorithm H is a heuristic for arbitrary task-processor configuration which uses a greedy type algorithm to reduce the task graph.

Similarly, Sarje [36] applied this simple model to a distributed system that shares a common bus or communication ring. A heuristic is presented to achieve optimum interprocessor cost under the load balancing constraints, which combines the clustering and reassignment.

In conclusion, graph theoretic models are simple, powerful when no or few constraints on the available system resources are imposed.

2.1.2 0-1 Integer Programming Model

The major drawback of graph theoretic models is that they cannot capture and satisfy the characteristics and requirements of complex distributed systems. Thus researchers have formulated the task allocation problem for distributed systems as an optimization problem and tried to solve it by using 0-1 integer programming models, which can easily introduce constraints of the systems as appropriate as possible.

The task allocation problem, formulated as a mathematical programming problem, can be stated in a general form as the problem of finding the values of variables X_{ik} which will minimize z, where

$$z = \sum_{i=1}^{N}\sum_{k=1}^{M} e_{ik}x_{ik} + \sum_{i=1}^{N-1}\sum_{j=i+1}^{N}\sum_{k=1}^{M}\sum_{q=1,q\neq k}^{M} c_{ij}x_{ik}x_{jq} \tag{1}$$

subject to

$$\sum_{k=1}^{M} x_{ik} = 1, i = 1, ..., N \tag{1.1}$$

$$\sum_{i=1}^{N} r_i x_{ik} \leq R_k, k = 1, \ldots, M \tag{1.2}$$

$$x_{ik} = \begin{cases} 1 & \text{if process } i \text{ on processor } k \\ 0 & \text{otherwise} \end{cases} \quad i = 1, \ldots, N, k = 1, \ldots, M \tag{1.3}$$

In the above formulation, objective function (1) consists of two parts. The first term is the sum of execution costs and the second term is the sum of the communication costs incurred between processes i and j residing on the different processors. In general, the communication costs between two processes residing on the same processor are negligible. Constraints (1.1) are imposed to ensure that every process i is allocated to a processor, and constraints (1.3) require the variables x_{ik} of the problem to be 0 or 1. Finally, constraints (1.2) refer to the resource limitations and ensure that the sum of resource requirements r_i of all processes i allocated to processor k does not exceed the processor's resource capacity R_k. Several sets of this type of constraints can be included with each referring to a different kind of resource. In the literature, different models have different sets of constraints for specific distributed systems [38].

The mathematical problem can be transferred to a linear one by introducing a set of assignment variables defined by $y_{ikjq} = x_{ik}x_{jq}$. The problem then becomes *minimizing* z, where

$$z = \sum_{i+1}^{N} \sum_{k=1}^{M} e_{ik}x_{ik} + \sum_{i=1}^{N-1} \sum_{j+i+1}^{N} \sum_{k=1}^{M} \sum_{q=1 q \neq k}^{M} c_{ij}y_{ikjq} \tag{2}$$

subject to constraints (1,1), (1,2), (1,3). Two more sets of constraints should be added to the model, in order to ensure the equivalence of the two formulations:

$$x_{ik} + x_{jq} - 1 \leq y_{ikjq} \text{ with } i \leq j, k \neq q \tag{2.1}$$

$$y_{ikjq} = 0, 1 \tag{2.2}$$

Although problem (2) is linear, the number of variables and constraints has been significantly increased. The increased dimensions of formulation (2) can be slightly reduced. Thus, the objective function (1) can be written as :

$$z = \sum_{i=1}^{N}\sum_{k=1}^{M} e_{ik}x_{ik} + \sum_{i=1}^{N-1}\sum_{j=i+1} c_{ij} - \sum_{i=1}^{N-1}\sum_{j=i+1}^{N}\sum_{k=1}^{M} c_{ij}x_{ik}x_{jk} \tag{3}$$

This can be linearized by introducing a set of slightly different assignment variables defined by $y'_{ijk} = x_{ik}x_{jk}$ yielding the following linear objective function *minimizing* z, where

$$z = \sum_{i=1}^{N}\sum_{k=1}^{M} e_{ik}x_{ik} + \sum_{i=1}^{N-1}\sum_{j=i+1} c_{ij} - \sum_{i=1}^{N-1}\sum_{j=i+1}^{N}\sum_{k=1}^{M} c_{ij}y'_{ijk} \tag{4}$$

subject to constraints (1,1), (1,2), and (1,3). Variable y'_{ijk} becomes one when both $x_{ik=1}$ and $x_{jk} = 1$. The additional constraints in this linearization are

$$x_{ik} + x_{jk} \geq 2y'_{ijk} \tag{4.1}$$

$$y'_{ijk} = 0, 1. \tag{4,2}$$

Therefore, nonlinear problem as (1) can be transferred to linearized one as (4).

Price [31] presented a model which is the same as the one mentioned above for a fully connected homogeneous system. No constraint is included in the model. An iterative technique that performs a series of transformations on an assignment matrix is presented. But it only guarantees the convergence to a locally optimum assignment. Global optimum can be achieved only when the communication costs is sufficiently small that they can be ignored.

Later, Price and Krishnaprasad [32] improved the model by introducing a constraint of limited memory for each processor. Three heuristics, – the iterative transformation algorithm, the clustering algorithm, and the Banded Q algorithm –, are presented. Performance analysis is reported which can serve as a guide to the practical utility of each method.

Chu [13] presented a model, where the objective function is modified as

$$Cost(X) = \sum_{k}\sum_{i}(e_{ik}x_{ik} + \sum_{l<k}\sum_{j<i} wv_{ij}d_{ki}x_{jl})$$

Similarly, the first summation term represents the processing cost for each module on its assigned processor. The second term sums the volume-distance products that represent the interprocessor communication costs (X is an assignment. v_{ij} denotes the volume of data sent from module i to module j, given by matrix V. d_{ij} denotes distance between processor i and j, given by matrix D.). Besides, Chu used a normalization constraint w to scale processing costs and interprocessor communication costs to account for any differences in measuring units. Two sets of constraints are introduced into Chu's model. A limited memory environment is represented by the constraint

$$\sum s_i x_{ik} \leq R_k \qquad k = 1, ..., N$$

where s_i represents the amount of memory storage required by module i and R_k represents the memory capacity at processor P_k. And the real-time constraint is given by

$$\sum u_i x_{ik} \leq T_k \qquad k = 1, ..., N$$

where u_i represents processing time required by module i and T_k represents the required time limit for processing the modules residing in processor k for a given task.

The model is solved by linearizing the objective function and adding further constraints. It is admitted that the dimension of the problem can be extremely large. A heuristic clustering algorithm is presented for this model by Chu. The systems considered here have heterogeneous processors but no replica of processes existing.

Gylys and Edward [21] presented a model similar to Chu's as mentioned above, except the objective function. They formulated the task allocation problem as a maximization problem where, instead of minimizing the interprocessor communication, it maximizes the intra-processor communication. Execution costs are not included in the model. In particular, the objective function in this model is represented

$$Q = \sum_{k=1}^{N} \sum_{j=1}^{M} \sum_{i=1}^{j} c_{ij} x_{ik} x_{jk}$$

which is the total amount of bus traffic eliminated (or intra-processor communication generated) by making program share the same processors. Two basic approaches for solving the optimal allocation problem were investigated. One is based on mathematical programming methods, the other on heuristic cluster analysis algorithm. This model is mainly suitable for the single bus systems. Other related work on the model can be further referred to [18].

Ma, Lee and Tsuchya [30] gave a more detailed and complete description of task allocation problem and the rules governing a distributed environment. The cost function is formulated to measure the interprocess communication costs and execution costs. Several constraints are used to satisfy different application requirements, including memory constraints, pre-allocation and co-location constraints, and constraints referring to replicate processes. The objective function used has the form *minimizing* z,

$$z = \sum_{i+1}^{N} \sum_{k=1}^{M} (we_{ik} x_{ik} + \sum_{j=1}^{N} \sum_{q=1}^{M} (c_{ij} d_{kq}) x_{ik} x_{jq})$$

where w is a normalization constant. This objective function is quite similar to formulation (1), but the interprocess communication costs between processes i and j depend not only on the relevant processes, but also on the distance (d_{kq}) between the processors to which they have been allocated. The problem of task allocation is tackled by a branch-and-bound exact method, and the solution procedure proposed consists of a search tree method.

As all the exact allocation algorithms were limited to very-small-sized problems, Billionet, Costa, and Sutter [4] applied the basic 0-1 integer programming model to a system in which, processors are heterogeneous, communication links are homogeneous, and the capacity of processors and links is unlimited, to study large scale problem. The branch-and-bound algorithm is applied using the proximate solution to the Lagrangean dual problem as the lower bound and using a zero duality gap to check consistency of 0-1 equation. Excellent experimental results are reported. Connolly [15] tried simulated annealing on a similar model. It performs extremely well, finding improved solution for several of the largest problems in the literature in only modest amounts of CPU time without the need to "tune" the system for each new data set.

In conclusion, the distinguish advantage of 0-1 integer programming model is that it can easily represent the distributed environment with as many constraints of the system as necessary. But the optimization problem of task allocation is more naturally expressible as a nonlinear programming problem. Most of them can be linearized as we stated previously, however, the resulting models are ill conditioned and their numerical stability can not be guaranteed.

2.1.3 Queueing Theory Model

Queueing theory models are usually used to model distributed processing systems. In particular, computers are represented as servers, modules' invocations as customers, and task invocations as external arrivals. Customers are routed for service in accordance with the task control-flow graph and the modules assignment. Each model incorporates a job routing strategy, which is divided into two classes: deterministic and nondeterministic.

For the heterogeneous system with nondeterministic routing (see Figure 2.3), each processor is modeled as a queue/server pair. Processor i is assumed to be characterized completely by its mean service rate μ_i, each job enters the system at mean rate λ. Their interarrival times are modeled as a sequence of independent identical distributed random variables. The distribution is assumed to be exponential with a mean of $\frac{1}{\lambda}$. An arriving job is routed to processor i with probability p_i ($p_i \neq 1$) where the routing decision is based on the outcome of an independent trial. Each processor services its queue of jobs in first-come-first-served priority order.

The processing time is modeled as an independent exponentially distributed random variable with mean $\frac{1}{\mu_i}$.

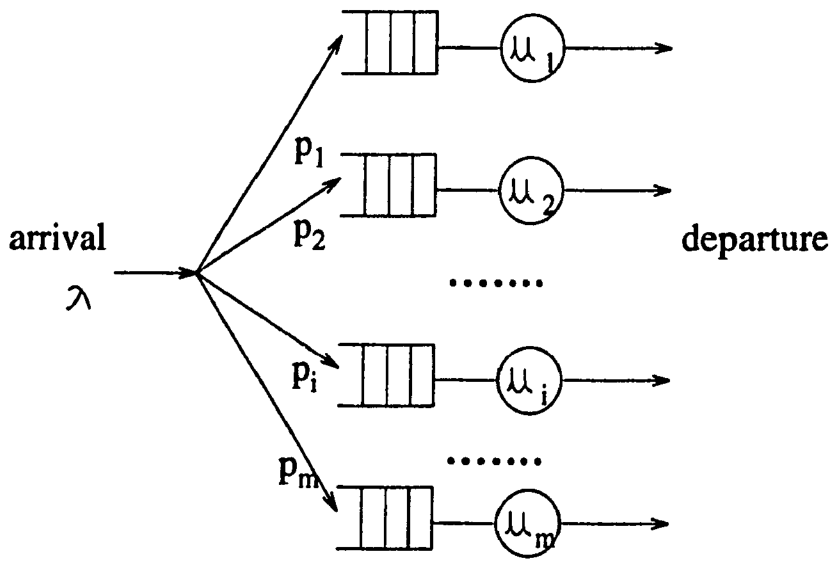

Figure 2.3: Heterogeneous system with nondeterministic routing

For the heterogeneous system with deterministic routing (see Figure 2.4), the model is the same as the previous one except the inclusion of job dispatcher. The job will be dispatched to queue $q(s, C)$, where $q(s, C)$ is determined by a system criterion function $C(n_1, n_2, ..., n_i, ..., n_m)$ which denotes the value of function when the job is sent to the *ith* queue. The job dispatching strategy can be set so that the next processor is chosen to minimize or maximize the expected value of a performance related criterion function.

Chow, and Kohler [11] presented three state-dependent routing policies: 1) the minimum response time policy, 2) the minimum system time policy, and 3) the maximum throughput policy. It is reported that the maximum throughput policy is conjectured to be optimal among all policies that are based on the system states, average arrival rates, and average service rate information. The model is proposed for heterogeneous system, but only load balance of the system is considered.

In [14], Chu criticized that a tractable queueing network model cannot represent

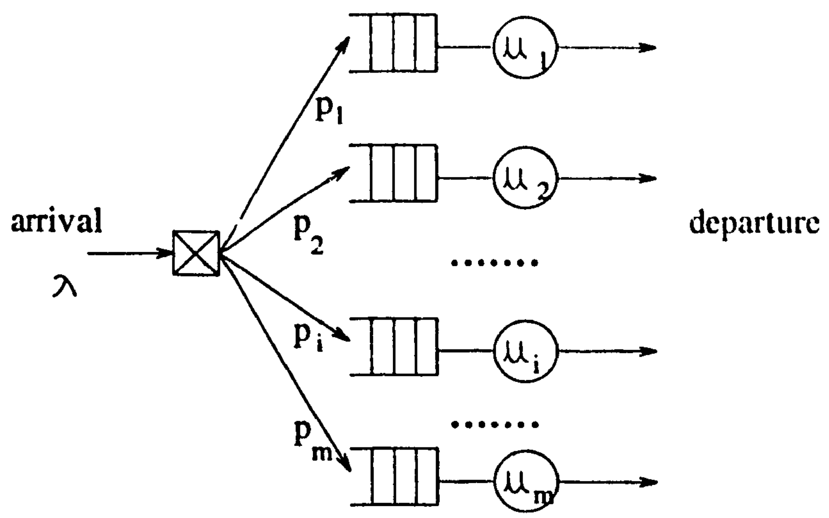

Figure 2.4: Heterogeneous system with deterministic routing

the distributed system because the routing policy in that model cannot represent the logical relationship among modules in the system. Therefore, he introduced an analytical model which consists of two submodels: the module response time model and the weighted control-flow graph model. It considers such factors as IPC, modules precedence relationships, module scheduling, interconnection network delay, and assignment of the modules and files to computers. Based on the model, they developed a search algorithm which uses the sum of task response time and delay penalty as the objective function. The algorithm searches for local optimal solutions and then selects the final solution from this set of local optimum. It is reported that the algorithm can generate the optimal solution for most cases by exhaustive searching. However, because of the exponential growth in computation requirements, for large-size system, such exhaustive search for optimal assignment is not feasible.

Queueing models are mainly used in dynamic task allocation. They can include many different factors of distributed system, but they are very sophisticated, and usually only load balancing can be considered. So far, research is mainly on small size systems.

2.1.4 Overview of Reviewed Models

Based on the study of the previous models in the literature, we can conclude that:

- Graph theoretic model is simple, powerful only when no or few constraints on the available system resources are imposed, it cannot capture and satisfy the characteristics and requirements of complex distributed systems.

- 0-1 integer programming model can easily represent the distributed system with as many constraints of the system as possible, but it can only solve linear problem. To be applicable to the nonlinear problem, it becomes ill conditioned and its numerical stability cannot be guaranteed. Unfortunately, the optimization problem of task allocation is more naturally expressible as a nonlinear programming problem.

- Queueing model is more suitable for dynamic task allocation. Basically, queueing model can include many different factors of a distributed system, but it is very sophisticated. Usually it only considers the load balance.

2.2 Models designed for our workstation farms

Although 0-1 integer programming model can include as many constraints as possible, it is only good at solving linear problem which is not natural for task assignment problem. On the other hand, Queueing model is more suitable for dynamic task assignment problem. Therefore, for task assignment problem on our workstation farms, we designed our models based on the theoretic graph model.

Two objective functions are often used in the literature. One is the *total cost* which is the total of the computation cost and communication cost incurred by the program on any processor; the other is *commpletion cost* which is the maximum cost including the computation cost and communication cost incurred by the program any processor. Using *completion cost* as an objective function for task assignment problems is more realistic, however, to calculate the completion cost for a task assignment itself will invoke a lot of calculations of maximization and minimization, for which we are short of efficient tools. Most of models in the literature use *total cost* as objective functions for task assignments. In this thesis, we use either of these two objective functions for different models.

We start with uniform m-way graph partition model for workstation farms with a single bus, then go to more complicated and realistic models. Neverthless, no constraints on system resources are considered in our models.

2.2.1 Uniform m-way Graph Partition Model

Considered a workstation farm in which all the processors are interconnected by a high-speed bus. We can model a parallel computation by an undirected graph $G = (V, E)$ in which each vertex in V represents a process and each edge in E represents a logical communication channel. The computation load of each process can be modeled by a function $w_1 : V \rightarrow I$ (I is the set of positive integers). The communication load of each channel can be modeled by a function $w_2 : E \rightarrow I$. Let $m > 0$ be the number of processors in a workstation farm. The task assignment problem for a workstation farm can thus be modeled by the following m-way graph partition problem: find a mapping $\pi : V \rightarrow \{1, 2, \ldots, m\}$ such that

$$W_2(\pi) = \sum_{\substack{e=\{u,v\}\in E \\ \pi(u)\neq\pi(v)}} w_2(e)$$

is minimized under the constraint that

$$W_1(\pi) = \sum_{1\leq i<j\leq m} |w_1(P_\pi(i)) - w_1(P_\pi(j))|$$

is minimal, where $P_\pi(i) = \{v \in V | \pi(v) = i\}$ for $1 \leq i \leq m$. (For any subset $C \subseteq V, w_1(C) = \sum_{v\in C} w_1(v)$.)

This model is designed for systems in which:

1. All processors are homogeneous;

2. All processors are connected by a high-speed bus;

3. There are no competition for system resources between any pair of processes assigned on the same processor;

4. Communication between any pair of processes within a processor is ignored.

2.2.2 Nonuniform Single-bus Total Cost Model

For a workstation farm of heterogeneous processors connected by a single high-speed bus, we can find that the performance of a program consisting of a set of modules depends heavily on the following three costs (times):

- *Computation costs.* Since the processors are heterogeneous, computation time of a task depends on the processor to which it is assigned. The computation load on the processor depends on the set of tasks assigned to it. To optimize the system performance, an assignment must attempt to balance the computation load of the program across the processors in the system.

- *Communication costs.* If two interacting tasks are assigned to different processors, the intermodular communications need to go through the interconnection network. The time for communication between two processors depends on the network topology, and the communication bandwidths of the links between processor pairs. The time spent by a processor on communication activity increases the completion time of that processor, and an assignment must keep these communication overheads to a minimum.

- *Interference costs.* If two tasks are assigned to the same processor, they will compete for the resources available on the processor (such as CPU, memory, I/O, etc.). This resource contention results in overheads, referred to as interference costs [28], which slow down the execution of either module, and reduce the processor untilization [3, 28]. These costs are processor dependent in a heterogeneous system. Since interference costs increase the execution time of the processor, an assignment must attempt to keep these overheads to a minimum.

Here we include the interference cost (introduced by Lo [29]) in the model to encourage parallelism in the system.

For the task assignment on this kind of workstation farms, we can formalize it as follows. For any integer $n > 0$, let $[n]$={1,2,3,...,n}, let $\Re$ be the set of all nonnegative real numbers. Assume that the system has $m > 0$ processors, and the program in the question has $n > 0$ task modules. Let $X : [n] \times [m] \to \Re$ be a function such that for any $i \in [n]$ and $j \in [m]$ $X(i,j)$ specifies the execution cost of running module i on processor j. Let C:$[n] \times [m] \to \Re$ be a function such that for any $i, j \in [n]$ $C(i,j)$ specifies the communication cost between modules i and j if they are assigned to different processors. Let $I : [n] \times [m] \to \Re$ be a function such that for any $i, j \in [n]$ $I(i,j)$ specifies the interference cost between modules i and j if they are assigned to the same processor. The task assignment problem then is to find a mapping $\pi : [n] \to [m]$ to minimize the *total cost*

$$cost(\pi) = \sum_{i=1}^{n} X(i, \pi(i))) + \sum_{\substack{\pi(i) \neq \pi(j) \\ i<j}} C(i,j) + \sum_{\substack{\pi(i) = \pi(j) \\ i<j}} I(i,j)$$

.

This mode is designed for systems in which:

1. All processors can be heterogeneous;

2. All processors are connected by a high-speed bus;

3. Communication between any pair of processes within a processor can be ignored.

In this model both the communication cost and the interference cost are processor independent.

Although using *total cost* as an objective function is simple and needs less CPU time for a task assignment, it is just approximate to completion cost. Some extensive expriments have been conducted to compare performance of models with the two ojective functions. Experiments for clusterd data sets, sparse data sets, and structured data sets are reported in Table 2.1 to Table 2.3. (Description of data sets and heuristics refer to Section 3.5)

	C30-3			C40-4			C50-5			C60-6		
	SP	SA	TA	SP	SA	TA	SP	SA	TA	SP	SA	TA
compl cost	976	976	994	1461	1501	1472	1965	1977	1976	2587	2590	2590
CPU (sec.)	0.2	3.3	6.7	1.6	8.8	4.1	6.1	14.8	42.4	7.0	8.7	10.3
total cost	1435	1435	1436	2342	2468	2342	3468	3643	3643	5152	5152	5226
compl cost	1407	1407	1407	2342	2468	2342	3468	3639	3643	5151	5151	5151
CPU (sec.)	0.1	0.9	0.2	0.1	1.4	0.1	0.1	1.1	0.2	0.1	1.6	0.3

Table 2.1: Performance comparisons for clustered data sets

	S30-3			S40-4			S50-5			S60-6		
	SP	SA	TA	SP	SA	TA	SP	SA	TA	SP	SA	TA
compl cost	248	252	254	277	280	284	350	350	352	461	473	464
CPU (sec.)	0.2	1.3	2.6	1.1	1.4	4.9	1.7	6.5	8.9	6.2	8.3	15.4
total cost	634	634	634	850	850	850	350	350	352	461	473	464
compl cost	267	300	300	305	305	305	387	411	406	504	555	554
CPU (sec.)	0.1	2.8	0.4	0.1	4.5	0.9	0.4	3.7	1.1	0.2	0.7	1.8

Table 2.2: Performance comparisons for sparse data sets

Each table consists of two regions for the performances of models with *completon cost* or *total cost* objectives. The up regions demostrate the costs of the task assignment with completion cost objective; while the down regions show the completion costs and CPU time of the task assignment and its final cost of the objective, i.e. the total cost.

	Line60-6			Ring60-6			Mesh49-6			Tree60-6		
	SP	SA	TA	SP	SA	TA	SP	SA	TA	SP	SA	TA
compl cost	200	202	208	435	437	437	180	180	194	196	199	231
CPU (sec.)	1.6	11.2	18.3	8.1	9.1	28.9	5.0	9.5	17.2	2.7	9.3	10.0
total cost	996	996	1013	1697	1709	1703	816	821	821	1002	1008	1011
compl cost	212	212	212	496	533	444	190	176	180	233	221	222
CPU (sec.)	0.1	1.2	0.7	0.5	1.7	2.0	0.1	1.2	0.7	0.5	3.5	2.6

Table 2.3: Performance comparisons for structured data sets

With the two regions in each table, we can compare the performances of the models using *total cost* and *completion cost* as objective functions. For an example, in Table 2.1 for cluster data set with $n = 30$ and $m = 3$ the assignment using *completion cost* for SP heuristic has completion time 1037. For the same data set, using SP heuristic with *total cost* objective results in an assignment with completion time 1352 (the entry corresponding to *compl cost*). This implies that the task assignment determined by model (using *total cost* objective) has a 35% increase in completion time over the assignment using *completion cost* objective model. Similiar conclusions can be drawn by observing the other entries in the tables; i.i. in each column we observe that *compl cost* using *total cost* objective is significantly higher than using *completion cost* objective.

Henceforth, for the same system, we also design the following model by using *completion cost* as an objective function.

2.2.3 Nonuniform Single-bus Completion Cost Model

Nonuniform Single-bus Completion Cost Model is designed for a similar workstation farm in Section 2.3.2, in which all processors are heterogeneous and are connected by a single high-speed bus. But this model tries to minimize the program completion time, which is more important in a parallel computing environment. We can formalize our task assignment on the workstation farm as follows:

For any integer $n > 0$, let $[n] = \{1, 2, 3, ..., n\}$, let $\Re$ be the set of all nonnegative real numbers. Assume that the system has $m > 0$ processors, and the program in question has $n > 0$ task modules. Let $X : [n] \times [m] \rightarrow \Re$ be a function such that for any $i \in [n]$ and $j \in [m]$ $X(i,j)$ specifies the execution cost of running module i on processor j. Let $C : [n] \times [m] \rightarrow \Re$ be a function such that for any $i, j \in [n]$ $C(i,j)$ specifies the communication cost between modules i and j if they are assigned to different processors. Let $I : [n] \times [m] \rightarrow \Re$ be a function such that for any $i, j \in [n]$ $I(i,j)$ specifies the interference cost between modules i and j if they are assigned to the same processor. The task assignment problem is then to find a mapping $\pi : [n] \rightarrow [m]$ to minimize the *completion cost*

$$cost(\pi) = \max_{1 \le k \le m} \{\sum_{i=1}^{n} X(i, \pi(i))) + \sum_{\substack{\pi(i) \neq \pi(j) \\ i<j}} C(i,j) + \sum_{\substack{\pi(i) = \pi(j) \\ i<j}} I(i,j)\}$$

This mode is designed for the system in which:

1. All processors can be heterogeneous;
2. All processors are connected by a high-speed bus;
3. Communication between any pair of processes within a processor can be ignored;
4. The system objective is to minimize program completion time.

In this model both the communication cost and the interference cost are processor independent.

2.2.4 General Nonuniform Total Cost Model

The target system of this model is connected by a interconnection network instead of a bus as in the previous two models. Processors in it are also heterogeneous.

Since the system consists of heterogeneous processors, *computation costs*, *communication costs*, and *interference costs* are therefore considered as its three basic costs to evaluate the performance of the system.

In this kind of workstation farm, the task assignment problem can be formally defined in graph theoretic terms as follows. For any integer $n > 0$, let $[n] = \{1, 2, 3, ..., n\}$. Assume that the system has $m > 0$ processors, and the program in question has $n > 0$ task modules. The distributed program can be represented as an undirected *task graph* $G = (V, E)$, where $|V| = n$, each vertex $v \in V$ represents a task module, and each edge $e = (u, v) \in E$ represents two communicating (interacting) modules. Each modules can be assigned to any one of the m processors. For any vertex $v \in V$ and any integer $1 \leq i \leq m$, $X(u, i)$ denotes the execution cost of running modules u on processor i. For any edge $(u, v) \in E$ and any integers $1 \leq i, j \leq m$, $C(i, j, u, v)$ denotes the *generic cost* between modules u and v if u is assigned to processor i and v is assigned to processor j. If $i \neq j$, $C(i, j, u, v)$ represents the interprocessor communication cost between modules u and v under this assignment. If $i = j$, $C(i, j, u, v)$ represents the interference cost between modules u and v caused by resource conflicts on the same processor.

Given a task graph $G = (V, E)$, a distributed system of m heterogeneous processors, along with the costs $X(u, i)$ and $C(i, j, u, v)$ for all $1 \leq i, j \leq m$ and $u, v \in V$, the task assignment problem is to find a mapping $\pi : V \longrightarrow [m]$ to minimize the *total cost*

$$cost(\pi) = \sum_{u \in V} X(u, \pi(u)) + \sum_{u \neq v} C(\pi(u), \pi(v), u, v)$$

Generally, the model is designed for system in which:

1. All processors can heterogeneous;

2. All processors are connected by an interconnection network;

3. Communication between any pair of processes within a processor can be ignored;

In this model both the communication cost and the interference cost are processor dependent.

For the same system, by using the *completion cost* objevtive, we design next model.

2.2.5 General Nonuniform Completion cost Model

The target system of this model is similar to the system in Section 2.3.4. However, the system objective in this model is to minimize program completion time.

First of all, we model the system's architecture interconnection as a graph $G_p = (V_p, E_P)$ where $V_P = \{p_1, p_2, ..., p_m\}$ denotes processors and E_p describes the interconnections among the processors. The properties of the interconnection network are provided by the delay matrix D, where $D(i,j)$ denotes the communication delay (cost/time) for sending a message (of unit length) from processor i to processor j, for $i,j \in \{1,2,...,m\}$. We assume that the communication delay between a pair of processors is identical in both directions, i.e., $D(i,j) = D(j,i)$.

For the parallel program (task), we can model it as a task graph $G = (V, E)$ where vertices $V = \{v_1, v_2, ..., v_n\}$ represents the interacting program modules (tasks) and the set of edges E represent data communication dependencies between the tasks. The characteristics of the program are represented by the following parameters:

- $X(i,k)$ represents the computational load of task i when executed on processor k, for all $1 \leq i \leq n$ and $1 \leq k \leq m$.

- $Y(i,j)$ represents the amount of communication required between tasks i and j, for $i,j \in \{1,2,...,n\}$. This is defined as the total number of unit length data to be transferred between the two tasks. Since we are assuming a blocking sending-receive model, we have $Y(i,j) = Y(j,i)$ for all $i,j \in \{1,2,...,n\}$. We have $Y(i,i) = 0$ for all $i \in \{1,2,...,n\}$. If there is no communication between tasks i and j then $Y(i,j) = 0$.

- $I(k,i,j)$ denotes the interference cost caused by assigning both modules i and j in processor k, for $i,j \in \{1,2,...,n\}$ and $k \in \{1,2,...,m\}$. As discussed in previous models, these costs reflect the overheads caused by both modules competing for the same resources. We assume $I(k,i,i) = 0$ for all $i \in \{1,2,...,n\}$.

Using the parameters Y and D we can construct an $m \times m \times n \times n$ *generic cost matrix* $C(i,j,u,v)$, for $i,j \in \{1,2,...,m\}$, $u,v \in \{1,2,...,n\}$, which can be computed as:

$$C(i,j,u,v) = \begin{cases} Y(u,v)\cdot D(i,j) & i \neq j;\ i,j \in \{1,2,...,m\};\ u,v \in \{1,2,...,n\}; \\ I(i,u,v) & i = j;\ i,j \in \{1,2,...,m\};\ u,v \in \{1,2,...,n\}; \end{cases}$$

If $i = j$, $C(i,j,u,v)$ denotes the interference cost on processor i when tasks u and v are executed on processor i. If $i \neq j$, $C(i,j,u,v)$ indicates the communication cost on processor i and j when communicating tasks u and v are executed on processor i and j respectively which is the time to send the message of size $Y(u,v)$ between processors i and j. Due to the blocking send-receive model, it follows that the cost function $C(i,j,u,v)$ is symmetric in terms of both parameter pair (u,v) and (i,j), i.e., $C(i,j,u,v) = C(j,i,v,u)$ for all $i,j \in [m], u,v \in [n]$.

Then our task assignment problem on this kind of workstation farm becomes to find a mapping π: $[n] \rightarrow [m]$ to minimize the *completion cost*

$$cost(\pi) = \max_{1 \leq k \leq m} \{ \sum_{\pi(u)=k} X(u,k) + \sum_{\substack{\pi(u)=k \\ \pi(u)\neq\pi(v)}} C(\pi(u),\pi(v),u,v) + \sum_{\substack{\pi(u)=k \\ \pi(v)=k}} C(\pi(u),\pi(v),u,v) \}$$

Generally, the model is designed for system in which:

1. All processors can be heterogeneous;

2. All processors are connected by an interconnection network;

3. Communication between any pair of processes within a processor is neglected;

4. The system objective is to minimize program completion time.

In this model both the communication cost and the interference cost are processor dependent.

Based on the different system configurations and the complexity of different objective functions we designed the above five models for task assignment on workstation farms. In the following chapters we design efficient heuristics for these models.

Chapter 3

Solution Techniques

In the review of task assignment models in chapter 2, we can see that the techniques used to tackle the task assignment problem in the literature can be classified into the following categories:

- Graph theoretic methods,
- Mathematical programming methods,
- Heuristic methods.

Graph theoretic algorithms are usually used for graph theoretic models such as Stone's max-flow/min-cut algorithm for the graph network flow model [40, 39], Shen and Tsai's state-space search A* algorithm for graph matching model [37], and so on. Mathematical programming techniques are usually proposed for the 0-1 integer programming models, such as the standard linear zero-one programming algorithm [12], branch and bound algorithm [30]. Standard queueing network techniques are often applied to the queueing models. All above techniques are proposed to achieve an optimal solution. However, obtaining the exact optimum to the task allocation problem for distributed systems, especially for systems with more than two processors, is often, in practice, either impossible or simply unattractive. In many formalizations of the task allocation problem, finding an optimal assignment of tasks to processors is found to be NP-hard in all but very restricted cases [17]. Therefore, much research focuses on the development of heuristic algorithms [13, 29]. Even though heuristic methods only aim to find suboptimal solutions, they are more practical, faster and simpler than the exact methods. In fact, in some cases, heuristic methods may be the only available tools for solving difficult problems. In the literature, heuristics have been extensively applied to different kinds of models for task assignment problem. For instance, Lo designed a quite successful heuristics for graph theoretic model [29], while Chu proposed a heuristic for his 0-1 integer programming model [13].

Tabu search and simulated annealing are two of the most important methods for general combinatorial optimization. Even though they are new (having histories less than 10 years) and still under development, they have claimed success in many application domains.

In this chapter, we first study these two methods, then we adapt them to our task assignment problem. To further reduce the execution time, we study the parallelism of these two approaches, and propose their parallel versions. However, with our experiments, we find that simulated annealing is too randomized while tabu search is too aggressive. Therefore, we develop our stochastic probe method which combines the advantages of these two methods. To further reduce the execution time, we also study the parallelism of this new method and propose its parallel version. More details are elaborated in the following sections. Before we go to theses heuristics, let us give some definitions (notations) which will make our discussion more conveniently and concisely.

3.1 Background and Notation

To describe the working of our following heuristics, we present the task assignment problem in the following forms, which is consistent with all of our five designed models.

$(p) \quad \textit{Minimizing } c(\pi) : \pi \in \Omega,$

where Ω is the set of all mappings $V \rightarrow \{1, 2, 3, ..., m\}$. The objective function $c(\pi)$ may be linear or nonlinear. In some setting (p) may represent a modified form of some original problems, as where π is a superset of the vectors that normally qualify as feasible, and $c(\pi)$ is a penalty function, designed to assure that optimal solutions to (p) likewise are optimal for the problem from which it is derived.

A wide range of heuristic algorithms for solving problems capable of being written in this form can be characterized conveniently by reference to sequences of *moves* that lead from one trial solution (selected $\pi \in X$) to another. Let S be the set of all defined moves. We use $S(\pi)$ $(\pi \in \Omega)$ to denote subset of moves in S applicable to π, and $S(\pi, v)$ $(\pi \in \Omega, v \in V)$ the subset of moves in $S(\pi)$ that redefines $\pi(v)$. For any $s \in S(\pi)$, $s(\pi)$, the new solution obtained by applying move s to π, is called a ***neighbor*** of π. We call $\{s(\pi) | s \in S\}$ the neighborhood of solution π in solution space Ω, and $|S(\pi)|$ the ***neighborhood size*** of solution π (all of our moves are 1-to-1 mappings). For any move s, we define the *gain* of s relative to the current assignment $\pi \in \Omega$ to be $g(s) = c(\pi) - c(s(\pi))$. Informally, given the current assignment π and the move s, function $g(s)$ returns the net improvement in the cost of assignment, obtained by applying s to π, over the old cost of π.

Vertex move and *vertex exchange* are two popular classes of moves for graph partition. Let $S_1 = \{(u, v) | \ u \in V, i \in [m]\}$ be the set of all moves for moving one module away from its current assigned processor. Given any move $s = (u, i) \in S_1$ and $\pi \in \Omega$, $s(\pi)$ is identical to π except that $s(\pi)(u) = i$. Let $S_2 = \{(u, v) | \ u, v \in [n]\}$ be the set of all modules swaps. Given any move $s = (u, v) \in S_2$ and $\pi \in \Omega$, $s(\pi)$ is identical to π except that $\pi(u)$ and $\pi(v)$ are swapped. Given any $\pi \in \Omega$, the neighborhood sizes of π based on S_1 and S_2 are $O(nm)$ and $O(n^2)$ respectively.

Our experiments show that module moves in S_1 are very effective in distributing the modules among the processors to minimize the total execution cost, while module swaps in S_2 are very effective in refining the assignment to minimize the total communication and interference costs. The best order and mixture of the module moves and module swaps are problem instance dependent.

To compromise the neighborhood size and the effectiveness of the moves, our algorithms use a special set S_3 of moves, where for any $\pi \in X$, $S_3(\pi) = S_1(\pi) \cup S_2'(\pi)$, and

$$S_2' = \{\text{exchange } u \text{ and } v \mid v \in V, \text{ moving } v \text{ to } P_\pi(j) \text{ maximizes gain which is} < 0;\ u \in P_\pi(j)\}.$$

Informally, we give module moves higher priority than module swaps. For a given assignment π and a given module v assigned to processor i, we first try to move v to all the other processors. If moving v to processor j has the best gain which is less than zero, then we also try to swap v with each of the modules assigned to processor j. Given any $\pi \in \Omega$, the neighborhood size of π based on S_3 is $O(nm)$. Experiments show that S_3 performs better than S_1 or S_2 alone in terms of both running time and solution quality for all our three task assignment heuristics [41].

These design issues addressed above are common to all the following heuristics.

3.2 Simulated annealing

Annealing is the physical process of heating up a solid until it melts and cooling it down until it crystallizes into a state with a perfect lattice. During this process, the free energy of the solid is minimized. But the cooling must be done very carefully so as to escape the local optimal lattice structures with crystal imperfections. In combinatorial optimization, there is a similar process which can be formulated as the problem of finding – among a potentially very large number of solutions – a solution with minimal cost.

Simulated annealing was first introduced by Kirkpatrick, Gelatt and Vecchi [23, 24] and Gerny [9] independently. It is based on a strong analogy between the physical annealing process of solid and the problem of solving large combinatorial optimization problem. It tries to avoid being trapped in local optima by accepting both "good" and "bad" moves at the beginning of the iterations, and gradually lowering the probability of accepting "bad" moves. It consists of iterative search of the solution space by repeating three steps:

1. Moving from the current solution to a new solution;
2. Evaluating the cost of the new solution;
3. Deciding to accept or reject the new solution to replace the current solution.

Even though in theory, simulated annealing can find global optima if we lower the above probability slowly in exponential time [41], its performance in a practical time frame depends heavily on the parameters comprising its "cooling schedule". In general, simulated annealing is time-consuming, but it has been successfully applied to many optimization problems.

3.2.1 Sequential simulated annealing

Simulated annealing can be viewed as an enhanced version of the local search. The central idea of simulated annealing is that some mechanisms are included to prevent an optimization scheme from getting stuck in a poor local optimum. At the very beginning, it attempts to accept both improving and worsening solutions. Little by little, the probability of accepting worsening solutions is reduced. This is done under the influence of a random number generator and a control parameter called *temperature* T which controls the probability of accepting the worsening solutions. As typically implemented [22], the simulated annealing approach involves a pair of nested loops and two additional parameters: a *cooling ratio* r, $0 < r < 1$, and an integer *temperature length* L. A generic simulated annealing algorithm is shown in Figure 3.1. In step 3 of the algorithm, the loop terminates when no further

```
1.  Get a random initial solution $\pi$.
2.  Get an initial temperature $T > 0$.
3.  While stop criterion not met do:
    3.1 Perform the following loop $L$ times:
        3.1.1 Let $\pi'$ be a random neighbor of $\pi$.
        3.1.2 Let $\Delta$ =cost($\pi'$)−cost($\pi$).
        3.1.3 If $\Delta \geq 0$ (uphill move),
                set $\pi = \pi'$.
        3.1.4 If $\Delta < 0$ (downhill move),
                set $\pi = \pi'$ with probability $e^{-\Delta/T}$.
    3.2 Set $T = rT$ (reduce temperature).
4.  Return the best $\pi$ visited.
```

Figure 3.1: Sequential Simulated annealing

improvement on $cost(\pi)$ seems likely.

There are two main issues related to the adaptation of this general approach to the task assignment problem. The first is the design of moves and the neighborhood structure, the other is the design of the cooling schedule which would mainly affect on the solution quality. We use S_3 (see Subsection 3.1) as the set of moves. More specially, during each iteration, we randomly choose two processors i and j ($i \neq j$), then we randomly choose a module u such that $\pi(u) = i$. If moving u to processor j has a nonnegative gain, then we use its resulting assignment as π'; otherwise we randomly choose a module v such that $\pi(v) = j$ and try to swap modules u and v, and use the assignment resulting from the move with better gain as π'.

As for the cooling schedule design, we made the following decisions.

1. We let $L = n \cdot$ SIZEFACTOR, where SIZEFACTOR is a parameter.

2. The initial temperature T_0 is chosen so that the initial acceptance rate is around *INITPROB*, another parameter in the range $(0, 1)$.

3. For each temperature, we measure the acceptance rate of the proposed moves. The algorithm stops when for five temperatures the acceptance rate is lower than MINPERCENT and the best visited solution is not improved in that

period of time. Here MINPERCENT is another parameter in the range (0, 1).

All the parameters for our simulated annealing algorithm are not independent. We tune the parameters of our annealing algorithm for each of our benchmark graphs one at a time. We repeat the process until no perturbation of the parameters can improve the performance.

The time complexity of each iteration of our simulated annealing heuristic is $O(n^2)$.

3.2.2 Analysis of parallelism for simulated annealing

Even though simulated annealing is generally applicable, flexible, and is theoretically guaranteed to converge to the global optimum, in practice the convergence procedure is extremely slow. It requires the potentially burdensome amount of time . This has motivated the development of parallel simulated annealing to reduce computation time.

Since each iteration of simulated annealing (including move, evaluate, and decide) depends on the result of the last iteration, simulated annealing has an inherent sequential nature in essence. In the literature, two basic approaches have been used to parallelize simulated annealing. In the first approach, the sequential decision making (Markov chains) are maintained; the resulting algorithms usually produce solution with qualities comparable to that of their sequential counterparts, but with a speedup within $O(\log P)$ for P processors [42, 25]. In the second approach the basic sequential properties of simulated annealing are broken in order to maximize speedup. The resulting algorithms usually have good speedup but produce inferior solution qualities to their sequential counterparts [8, 16, 1].

Kravitz and Rutenbar proposed a parallel simulated annealing with the serial decision-making sequence maintained. As stated in [42], on four processors, only speedup 2.5 is obtained for the placement problem, and additional processors would not improve the run-time significantly.

In [42], by using speculative computation, Witte, Chamberlain and Franklin developed another parallel simulated annealing which is problem-independent and maintains the serial decision sequences. In essence, they make use of the enough number of processors in the system to build a processor tree to speculate about the total number of iterations required at a temperature. They assign one processor to work on the first iteration and use two other processors to perform speculative computation. One of the other two processors speculates that the result will be an accept and begins to work on the next iteration under this assumption; while the other processor speculates that the result will be a reject. When the decision is made, some work has been done on next iteration by the other two processors. The three processors make a binary tree in which the first processor is the root and the other two processors are the left (accept) and the right (reject) children of the root. To begin work on the next iteration, the root must send the accept or the reject processor a current solution. Since the reject processor assume the root will retain its current solution and thus needs the current solution. The accept processor assumes the the root will replace the current solution with the new solution and thus needs the new solution generated at the root by the move function. The root determines which of the two child processors has correctly speculated about the outcome of the first iteration. Once the decision is made, the solution chosen by the incorrectly speculating child will be discarded. The binary tree can be extended another level with 2 exponential number of processors which speculate the outcome of the subsequent iteration. This can continue until no processor is available.

However, in their work, only $\log_2 P$ speedup on P processors is reported by this parallel simulated annealing. This is because of some potential problem in this parallelization. In general, there will not be enough processors to speculate about the total number of iterations required at a temperature. As a consequence, it needs to take out some processors from the existing tree for the deeper levels. Therefore, it needs a lot of time to layout the processors to construct the tree. On the other hand, when the tree goes to deeper and deeper levels, the serial decision sequence is destroyed little by little.

From either practical or theoretical point of view, we can see that to maintain the serial decision sequence will sacrifice a significant part of execution time which will hamper us from getting a good speedup. Therefore, our consideration is placed on the second class which violates the serial sequence somehow but still can achieve solution with quality comparable to that of sequential version. Some parallel simulated annealing algorithms of this class are proposed in the literature [8, 16, 1]. However, they are tailed to certain specific problems. Our work concerns the development of a parallel simulated annealing which can be generally applied to different problems.

Two considerations advise our development of parallel simulated annealing:

1. In theory, an execution of simulated annealing is a Markov chain which consists of certain number of Markov subchains , and each Markov subchain corresponds to a part of execution at a certain temperature[2]. Therefore, to achieve a solution with good quality, we should maintain the chain structure (serial decision sequence) somehow.

2. Basically, simulated annealing is a search of solution space. At each temperature, each Markov subchain searches a subset of the solution space. By expanding the set of solution space searched by each Markov subchain with multiple processors, it can search the same size of the solution space with less time.

Based on these two considerations, we develop two strategies to parallelize simulated annealing:

1. **Reduce the whole length of Markov chain for an execution of simu-**

lated annealing

At each temperature, we use n processors to accomplish n Markov sunchains simultaneously (independently or interactively). To put it another way, Step 3.1 in Figure 3.1 is executed by n processors in parallel. Ideally, the solution space searched by n processors at a temperature is n times the space searched by one processor. With this assumption, we thus reduce the whole length of the Markov chain to $1/n$, which will lead to the same solution space size searched by a sequential simulated annealing.

2. **Maintain the whole structure of Markov chain for an execution of simulated annealing but reduce the length of each Markov subchain**

   As shown in Figure 3.1, we use p processors to execute $1/p$ part of the loop 3.1 simultaneously. More details are discussed in Section 3.3.

Extensive experiments are conducted for these two parallelization strategies. Our experiments show that the first strategy for parallelization of simulated annealing cannot obtain a good speedup even with 2 or 3 processors while the second strategy can achieve a much better speedup with up to 10 processors with solution quality comparable to those of its sequential version.

After studying more deeply on the first strategy, we find that $1/n$ length of the Markov chain does not mean that it can be executed with $1/n$ execution time. Because during each execution of simulated annealing, it spends much more time on each high temperature than on each low temperature. In other words, the execution time of simulated annealing is dominated by the execution at high temperatures (initial cooling period). That is why we cannot obtain speedup by reducing the cooling period. Figure 3.2 shows the execution time distribution over different temperatures during cooling procedure for Graph $R100\_2$.

Our second strategy basically maintains the inherent nature of simulated anneal-

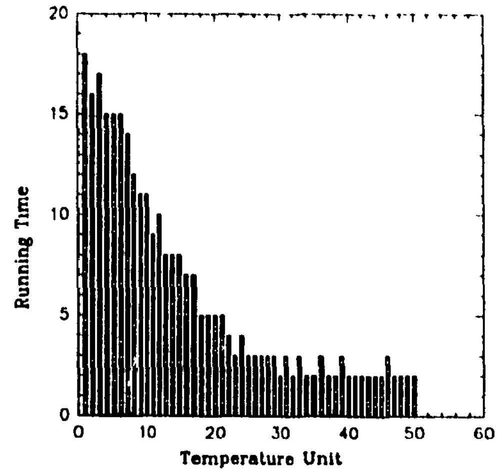

Figure 3.2: Execution time distribution over temperature units for Graph $R100\_2$

ing, the serial decision sequence or the Markov chain. It only changes the Markov subchains somehow by using multiprocessors. We apply this strategy to parallelizing the simulated annealing.

Even though we have not got speedup from the first parallelization, we believe that there is still some potential for this strategy. Cooling schedule here is a critical factor which might improve the result. Here we propose some strategies for the cooling schedule as future work:

- Speed up the cooling procedure;
- Find out a proper final temperature to terminate the execution, which will not terminate the execution at the initial phase but still allow a good speedup.

3.2.3 Parallel simulated annealing

To compromise solution quality and speedup, we adopt the second approach and designed the parallel simulated annealing algorithm outlined in Figure 3.3 for our task assignment problem.

Our parallel simulated annealing uses the same cooling schedule as sequential one. Temperature decreases by the factor r. But at each temperature, n processors

```
1.   Get a random initial public solution $\pi_p$.
2.   Get an initial temperature $T > 0$.
3.   Set the sequential loop length $L_s$;
     Get the parallel loop length $L_p = L_s/n$; (n:processor number)
4.   While stop criterion is not met do:
     4.1  Copy solution $\pi_p$ to the local solution $\pi_l$.
     4.2  Perform the following loop $L_p$ times:
          4.2.1  Let $\pi_l'$ be a random neighbor of $\pi_l$.
          4.2.2  Let $\Delta = W_3(\pi_l') - W_3(\pi_l)$.
          4.2.3  If $\Delta \geq 0$ (uphill move),
                    set $\pi_l = \pi_l'$.
          4.2.3  If $\Delta < 0$ (downhill move),
                    set $\pi_l = \pi_l'$ with probability $e^{\Delta/T}$.
     4.3  Update $\pi_p$ with the best $\pi_l$ visited in the last
          run of step 4.2
     4.4  Set $T = rT$ (reduce temperature).
5.   Return $\pi_p$.
```

Figure 3.3: Parallel Simulated annealing

do step 4.1 in parallel. The algorithm starts with an initial public solution π_p. At the same initial temperature T, each processor copies the public solution π_p to its local version π_l, does loop L_p times to find the best local solution π_l', then at the end of the execution of step 4.1, the best solution π_l found so far within the last L_p iterations updates the π_p if it is better than the current π_p. Then the temperature is reduced by step 4.3, the algorithm goes to the next phase of search with the new temperature. The algorithm terminates until some stop criterion is met.

During each temperature, we reduce the loop length L_s to L_p ($L_p = L_s/n$) for each processor's execution. Since we use n processors to do the loop 4.1 simultaneously (each processor searches its own set of solution space), ideally, the size of the set of solution space searched by the parallel simulated annealing at each temperature should be almost the same as the one searched by sequential solution. However, in practice, the sets of solution space reached by different processors overlap to some extent. In other words, the size of the solution space set reached at certain temperature is smaller than the one searched by sequential algorithm. To make the possibilities of the overlap as little as possible, we try to relate the random number generator at step 4.1.2 to the executing processor's ID in the system.

Tuning parameters in the parallel simulated annealing are the same as those in sequential version.

Experiments show that our parallel simulated annealing algorithm can produce solutions with qualities comparable to those of its sequential counterpart, and achieve a speedup better than those reported in the literature with similar solution qualities.

3.3 Tabu search

Tabu search is a new approach to combinatorial optimization characterized by aggressive local search during each iteration, and avoiding cycling in the solution space by keeping a short history of the attributes of the recent moves [19, 20]. It differs from simulated annealing on two main aspects:

- It is more aggressive. For each iteration the whole neighborhood of the current solution is usually searched exhaustively to find the best candidate moves.

- It is deterministic. Each iteration repeats the above exhaustive search for best candidate moves. The best candidate move which does not cause cycling in the solution space will be used no matter what sign its gain has. A *tabu list* is usually used to record the recent move history to avoid solution cycling, so comes the name of the approach.

In general, tabu search algorithms are slower than other problem-specific heuristics, but they have achieved impressive success in many problem domains.

3.3.1 Sequential tabu search

Tabu search is distinguished by two key elements [19, 20]:

1. Constraining the search by classifying certain of its moves as forbidden (tabu);

2. Freeing the search by a short term memory function that provides "strategic forgetting."

Figure 3.4 outlines a generic tabu search algorithm using π to represent a solution, *cost* the cost function, and t the length of the tabu list. Given a random solution,

```
1.   Get a random initial solution $\pi$.
2.   While stop criterion not met do:
     2.1  Search whole neighborhood of $\pi$, get a
          neighbor $\pi'$ maximizing
          $\Delta = cost(\pi') - cost(\pi)$ and not visited
          in the last $t$ iterations.
     2.2  Set $\pi = \pi'$.
3.   Return the best $\pi$ visited.
```

Figure 3.4: Sequential Tabu search

the algorithm repeats the loop at Step 2 until some stop criterion is met. During each iteration, the algorithm makes an exhaustive search of the solutions in the neighborhood of the current solution which has not been traversed in the last t ($t > 1$) iterations. The best solution found in this process will be used to replace the current solution.

The main design issues for a tabu search heuristic are as follows:

1. The design of the neighborhood (moves) of the current solution. A large neighborhood usually makes each iteration more aggressive but also more time-consuming.

2. The design of the contents of the tabu list. If move s is used to transform the current solution to π, the corresponding cell of the tabu list should capture some attributes of π or s so that π will not be traversed again in the next t steps. At one extreme, we can store solution π directly in the tabu list. But in practice, to save memory and checking time, some attributes of s will be stored in the tabu list to prevent s to s^{-1} (resvered s) to be used in the next t iterations.

If we use a more detail set of attributes of a solution or a move in each cell of the tabu list, more memory space and checking time will be incurred during the solution-space search, and the searches will be less restrictive since less solutions (in addition to the ones visited in the last t iterations) will be tabued. On the other hand, if we use a more abstract (simplified) set of attributes of a solution or a move in each cell of the tabu list, the implementation will be more space and time efficient for each iteration, and the searches will be more restrictive since more extra solution will be tabued.

3. The design of the aspiration level function. To make the implementation more space and time efficient, most designs of the contents of the tabu list will have too many solutions in addition to those visited in the last t iterations, thus risk to lose good move candidates. As a make-up, we can define an aspiration level $A(s, \pi)$ (usually an integer) for each pair of move s and solution π such that if $cost_1(s(\pi)) < A(s, \pi)$ the tabu status of s for the current solution π can be overridden. In practice some attributes of π, instead of π itself, will be used in the definition of $A(s, \pi)$. $A(s, \pi)$ is designed to capture the common properties of the earlier applications of s to solutions sharing the same attribute values as π.

4. The design of the length t of the tabu list. Parameter t determines how long the move history will be saved in the tabu list. Suppose that π is a local optimum, and it needs at least t' consecutive "uphill" moves to go to another local optimum π'. Then $t \leq t'$ is a necessary condition for π to reach π'. In general, the longer the tabu list is, the more time it spends on checking tabu status for each move, and the more restrictive the search process is. On the other hand, a too short tabu list risks introducing cycling in the solution space. Parameter t can be a constant or a variable during the execution of the heuristic. For many applications, a tabu list length around 7 is found appropriate [19].

The following is the description of our tabu search heuristic for task assignment:

1. We use S_3 in Section 3.1 to define the moves and the neighborhood of the current solution.

2. For the tabu list design, we use a circular list to maintain the vertices moved (swapped) in the last t ($t > 1$) iterations. We find that a more detailed characterization of the past moves usually traps the search process in a small subspace of the solution space (many vertices may never be moved). A constant tabu list length of 5 produces the best performance for most of our problem instances.

3. We use the cost of the best visited solution as the aspiration level $A(s, \pi)$ for all pairs of s and π. Based on the same observation pointed out in the last item, more "flexible" searches implemented by a more sophisticated aspiration level definition tend to limit the real search freedom in the solution space.

The time complexity of each iteration of our tabu search heuristic is $O(n^4)$.

3.3.2 Parallelization of tabu search

Unlike simulated annealing, tabu search performs the transition from one feasible solution to another deterministically. In each iteration, the whole neighborhood is searched before the best solution (according to a given criterion) is found and taken as the current solution. Empirical studies show that tabu search is slower than problem-specific heuristic, it spends excessive running time on aggressive search.

At present, few parallelization efforts are reported in the literature. Roberto Battiti and Giampietro Tecchiolli proposed a parallel tabu search which basically let each processor execute the sequential tabu search algorithm independently and report the best solution [3].

For tabu search heuristic on our task assignment problem, we observe that the execution of tabu search can be functionally divided into two parts, which contain quite different features for parallelization:

Part A: the part of the aggressive searches (composed of all search for the best neighbor at each iteration). It takes over 90% of the running time, and is executed independently without too much information exchanged.

Part B: the part to administrate the execution procedure. It needs to access a lot of information.

From parallelization point of view, it is obvious that Part A should be parallelized, because it does the mechanical search and needs little information. In theory, we can derive ideal speedup with a certain *parallelized rate*[1]. For example, if the parallelized rate is 90%, then

$$speedup = \frac{T_0}{0.1 \times T_0 + 0.9 \times T_0/n}$$

where T_0 be the running time of the algorithm executed by one processor, and n be the number of processors involved.

Figure 3.5 gives an overview of the speedup in theory. Each curve corresponds to the speedup for different parallelized rate with up to 10 processors. From this figure, we can conclude that a good speedup can be obtained by parallelizing the **Part A** when it takes over 90% running time of the whole program.

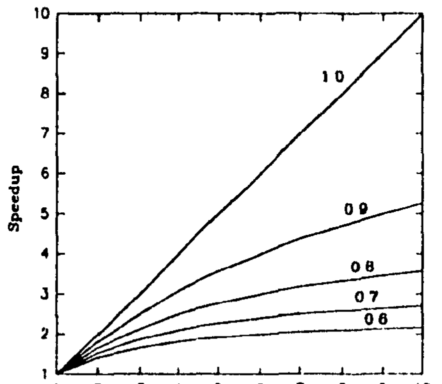

Figure 3.5: Speedup for different parallelization rate

[1]The percentage of the part of a program being parallelized over the whole program.

Our parallel tabu search algorithm outlined in Figure 3.6 is based on the even partition of the solution search space during each sequential iteration.

To evenly divide the solution search space among the processors, we divide the whole neighbourhood (at a certain step) into n parts with each part containing approximate $1/n$ neighbours (n is the number of processors available in the system). In particular, for a system with m processes and n processors, the whole size of its neighbourhood is $mn/2$. The neighbourhood can be defined as a search area determined by two interger variables i and j with $1 \leq i \leq n$ and $1 \leq j \leq m$ and $j \leq i$. Then at that certain step, the part of nerighbourhood assigned to procesor k is an area with $j : 1 .. i$ and

$i : 1 .. \left\lfloor \frac{n}{\sqrt{m}} \right\rfloor$ when $k = 1$, i.e. for the first processor;

or

$i : (\left\lfloor \frac{n\sqrt{k-1}}{\sqrt{m}} \right\rfloor + 1) .. \left\lfloor \frac{n\sqrt{k}}{\sqrt{m}} \right\rfloor$ when $1 < k < m$;

or

$i : (\left\lfloor \frac{n\sqrt{n-1}}{\sqrt{m}} \right\rfloor + 1) .. \left\lfloor \frac{n\sqrt{n}}{\sqrt{m}} \right\rfloor$ when $k = n$, i.e. for the last processor.

Therefore, each processor in the system will get a evenly n-divided part of the neighbourhood at that step.

```
1.  Get an initial public solution $\pi_p$ randomly.
2.  Set a strategy to partition the neighborhood of a solution
    during each iteration
3.  While stop criterion not met, do in parallel:
    3.1  Copy $\pi_p$ to local solution $\pi_l$.
    3.2  Search the corresponding part of the neighborhood
         of $\pi_l$, get a neighbor $\pi_l'$ maximizing
         $\Delta = cost(\pi_l') - cost(\pi_l)$ and not visited
         in the last $t$ iterations.
    3.3  Update $\pi_p$ to $\pi_l'$ if necessary.
4.  Return the best $\pi_p$ visited.
```

Figure 3.6: Parallel Tabu search

Parallel tabu search starts with a random initial solution. The neighborhood of the current solution during each iteration is evenly partitioned among the processors. During each iteration, each processor first copies the public solution to its local one, then searches its corresponding part of the neighborhood to find a best neighbor in this neighborhood which has not been traversed in the last t ($t > 1$) iterations, and updates the current public solution π_p with the best neighbor found by the processor if necessary. The algorithm repeats the loop at Step 3 until some stop criterion is met.

Tuning parameters in the parallel tabu search algorithm are the same as those in its sequential version.

Experimental study shows that this parallel tabu search algorithm can achieve almost linear speedup for large problem instances.

3.4 Stochastic probe heuristic

In general, simulated annealing and tabu search heuristics are slower than problem-specific heuristics. Their excessive running time mainly results from the search strategies of these two heuristics.

- As for the simulated annealing algorithm, it is not aggressive in neighborhood search. Each iteration chooses a random neighboring solution, which is usually not the most profitable one. The solution cost improves mostly in a narrow time range. The solution searches after this range is mainly limited to a small subspace of X.

- As for the tabu search algorithm, the utilization of information is low. For example, if we use S_3 to define the moves, then each iteration needs to search a neighborhood of roughly $n + \frac{n^2}{2m}$ solutions while using the information for only few (no more than the length of the tabu list plus one) of the neighboring solutions. The deterministic search process also limits the solution search to a small subspace of X, as evidenced by the fact that many vertices are never moved during the execution [41].

The objective of this section is to introduce a new approach for general combinatorial optimization.

3.4.1 Stochastic probe heuristic

Stochastic probe heuristic design is based on our following convictions:

- Aggressive neighborhood searches are essential to finding "good" solutions in a practical time frame. But a more aggressive search usually implies more search time. While tabu search and simulated annealing approaches represent the two extremes, a good trade-off must be made to compromise the aggressiveness and the running time of the search process.
- A good search algorithm should have the ability to effectively leave local optima when they are reached. The trace of the current solution should be controlled by the recent move history, not by "random walk."
- Randomized search is more effective in avoiding cycling in solution space than the tabu list technique. But the acceptance of moves with very bad gains (as simulated annealing does in high temperature) is usually not profitable.

The result is a combination of the aggressive search process in the tabu search and the stochastic search process in the simulated annealing approach. We call our new approach *stochastic probe.*

Given an initial solution π and a vertex v, we use $S(\pi, v)$ to denote the subset of movies in $S(\pi)$ that redefines $\pi(v)$. For any integer $p \leq 0$, we use $random(-p)$ to represent a random integer between $-p$ and 0 inclusively. Figure 3.7 outlines our stochastic probe approach. Informally, the algorithm consists of a sequence of

```
1.  Get a random initial solution $\pi$.
2.  Let $L$ be a circular list of the vertices in $V$.
    Set $v$ to any of the vertices in $V$ (the current vertex).
3.  While stop criterion not met do:
    3.1  While there is any $\Delta > 0$ in the last $k$ iterations of this loop do:
         3.1.1  Let $v$ be the next vertex down the list $L$.
         3.1.2  Let $s \in S(\pi, v)$ maximizing $\Delta$ in 3.1.3.
         3.1.3  Let $\pi' = s(\pi)$, $\Delta = cost(\pi') - cost(\pi)$.
         3.1.4  If $\Delta > random(-p)$, set $\pi = \pi'$.
    3.2  Set $\beta$ according to current statistics.
    3.3  Perturb randomly the value of $\pi(u)$ for $\beta$% of the vertices $u$ in $V$.
4.  Return the best $\pi$ visited.
```

Figure 3.7: Stochastic probe

well-organized probes, each probe searches for a local optimum. The last solution in a probe will be modified randomly to some extent so that it becomes the initial solution for the next probe. The algorithm stops when no improvements on the best visited solution occur for several consecutive probes. Each probe in turn consists of a sequence of iterations; each iteration makes an aggressive search for the most profitable move involving the current vertex. Variable p is used to control the tolerance of "bad" moves. The chosen move will be accepted if and only if it has a gain greater than random(-p). This mechanism is designed to help the solution search leave local optima when they are reached. A probe finishes when the gains for the last k consecutive iterations are all less than or equal to zero. The following are the main design issues to apply this approach to the solution to a particular problem.

- The parameter p. The value of p controls the extent of tolerance for "bad" moves.

- The parameter β. A small β will lead to more thorough solution searches in a small subspace of ω, whereas a large β will enlarge the search range to exploit more local optima.

- The stop criteria for each probe and for the heuristic. The former is determined by k. A larger k makes a more thorough probe into a subspace of ω with more running time. There are similar trade-offs for the stop criterion of the heuristic.

For the task assignment problem, we find that the following decisions are appropriate:

- We run the algorithm for 1000 iterations with p set to zero. We set p to 20% of the average absolute value of the negative gains for the 1000 iterations.
- We set β to a fixed value ranging from $10n$ to $20n$ depending on problem instance.
- We set k to a value from $0.2n$ to $0.9n$ depending on problem instance.
- The algorithm stops when the best visited solution cannot be improved for a few consecutive probes.

The time complexity of each iteration of our stochastic probe heuristic is $O(n^3/m)$.

3.4.2 Analysis of parallelism for stochastic probe

Our stochastic probe is a generally applicable optimization approach based on iterative search techniques, which takes the advantages of both the randomized search in simulated annealing and aggressive search in tabu search. Since our problem is task assignment on workstation farms, we should not only solve the problem as fast as possible but also take the advantages of the existing parallel computing facility supplied by the workstation farms themselves. Therefore, we further study the parallelism of stochastic probe. Our design of the parallelization focus on the following two aspects:

- Improving the solution quality with comparable execution time;

- Reducing the execution time with comparable solution quality, i.e. obtaining significant speedup;

Sequential stochastic probe heuristic consists of a serial of well-organized probes. An execution of stochastic probe algorithm is a search of solution space, and an execution of a probe is a search of solution subspace. Each probe starts with a randomly modified solution from the last probe. By expanding each probe's search subspace, it will need less number of probes for same quality solution, or same number of probes for the better quality solutions.

This conviction motivates the design of parallelizing our stochastic probe algorithm. Two strategies have been designed and studied.

- **For better quality solution with comparable execution time**

  As discussed in Section 3.3.2, in sequential stochastic probe, each probe starts with a solution modified from the last one, then randomly chooses a vertex u. It first tries to find a partition p which will promise best cost gain $G_1$by moving vertex u to partition p. If $G_1 \leq 0$, it tries to find a vertex v in partition p which will promise best cost gain G_2. An action is taken by following the rules:

  - move vertex u to partition p if $(G_1 > 0)$ or $(G_1 \geq G_2)$;
  - exchange vertex u with vertex v if $(G_1 < 0)$ and $(G_1 < G_2)$.

  In general, this procedure is to find a neighbor, which is the best one based on the randomly chosen vertex v. Therefore, with n processors available, we can get n neighbors based on n randomly chosen vertices, and choose the best one. As a consequence, the algorithm can search more thoroughly at each step. To an extreme, by using enough processors, a one-step search can be a whole neighborhood search at that step.

- **For less execution time with comparable quality solution**

  With multiple processors available, each probe's searching space is expanded by multiple processors' concurrent execution. Each processor independently executes its own probe search with its initial solution, which is modified somehow from a public solution. Once all processors finish their probe search, the best solution is chosen as the final (best) solution of this probe.

Extensive experiments are conducted on both of the two parallelization strategies. We apply the first strategy of parallelization on our five models.

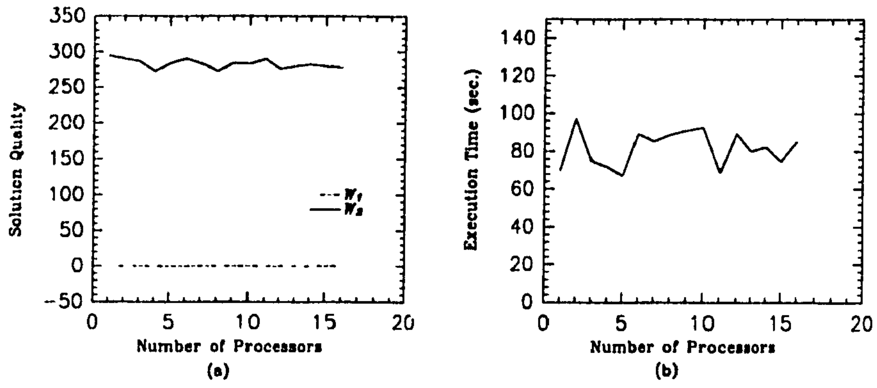

Figure 3.8: Graph $R200\_4$ for m-way graph partition model (n=200, m=4) (a) Solution quality; (b) Execution time.

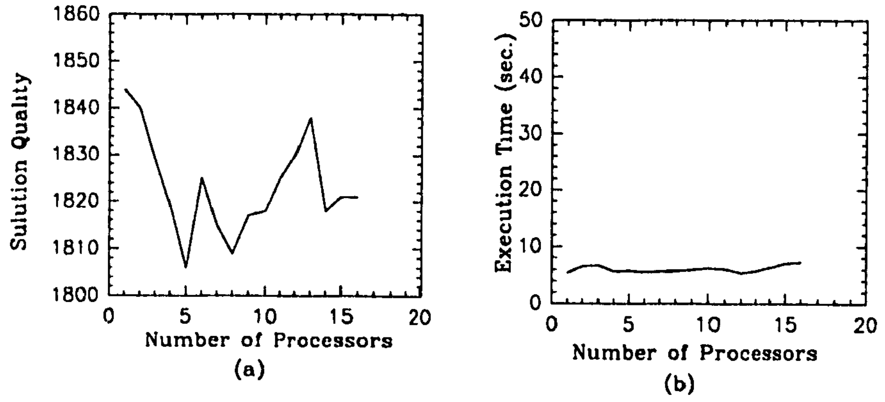

Figure 3.9: Graph $S60\_6$ for uniform total cost model (a) Solution quality; (b) Execution time.

Figure 3.8 to Figure 3.12 show the performance regarding solution quality on example graphs, one for each of the five models. (More detail on these graphs can

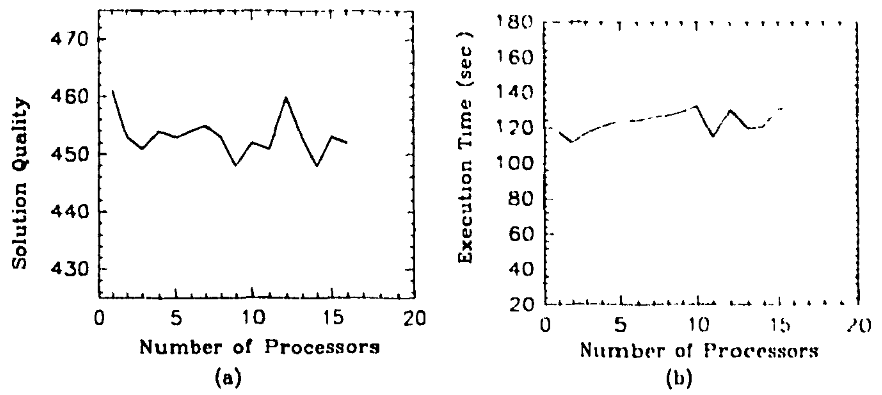

Figure 3.10: Graph $S60\_6$ for uniform completion cost model (a) Solution quality; (b) Execution time.

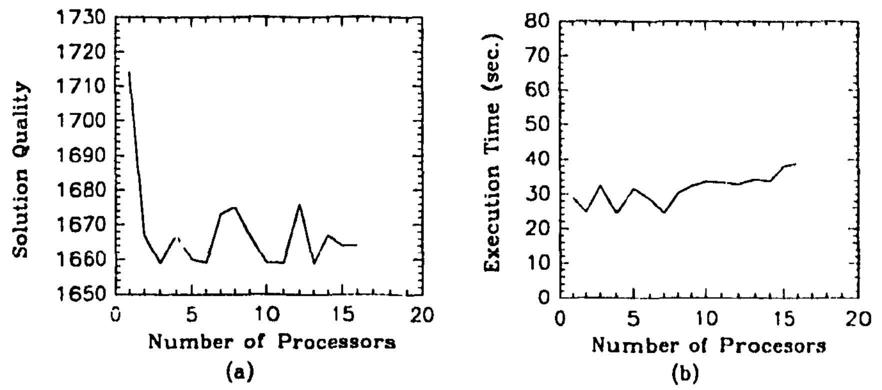

Figure 3.11: Graph $S60\_6$ for nonuniform total cost model (a) Solution quality; (b) Execution time.

be found in next section.)

Experimental results show that quality improves when more processors involves, but not satisfactorily. This is due to the fact that combinatorial optimization is in general very complicated, a current best solution at first step does not always lead to a final optimal solution. There should be a mechanism such as a function which can direct the searching procedure. Unfortunately, it is usually time consuming as studied in the literature.

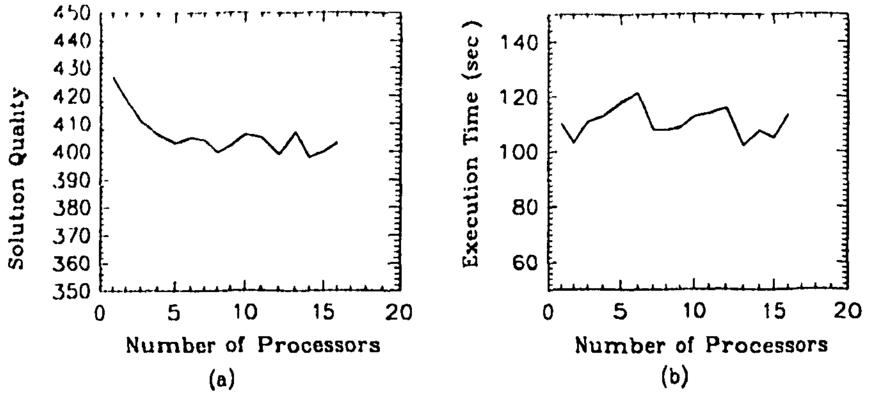

Figure 3.12: Graph $S60\_6$ for uniform completion cost model (a) Solution quality; (b) Execution time.

For our task assignment problem, we focus more on reducing execution time. Our experiments demonstrate that the second strategy can obtain a desirable speedup on all of our five models. Therefore, we apply this strategy to the parallelization of stochastic probe for task assignment. More details are discussed in the Section 3.4.3.

3.4.3 Parallel stochastic probe

As analyzed in Subsection 3.3.2, our parallel stochastic probe algorithm is based on the idea of expanding each probe's searching subspace. we parallelize it by reorganizing its probes as outlined in Figure 3.13. The heuristic consists of a sequence of searching phases. Each searching phase consists of a certain number of well-organized probes (K_1 probes). Each searching phase tries to find an optimum in separate solution subspace, and each probe tries to find a local optimum as probe in sequential version does.

During each searching phase, each processor performs the following operations in parallel: (1) get a local copy of π_p, and randomly modify it using parameter β_1 to get its local initial solution π_l; (2) run the sequential stochastic probe; (3) use the best solution visited to update the global solution π_p. The searching phases are repeated until the stop criterion is met.

```
1. Generate a random initial public solution $\pi_p$.
2. Let $L$ be a circular list of the vertices in $V$.
   Set $v$ to any of the vertices in $V$ (the current vertex).
3. While stop criterion not met, do in parallel:
   3.1 Copy the public solution $\pi_p$ to the local solution $\pi_l$.
   3.2 Perturb randomly the value of $\pi_l(u)$ for $\beta_1\%$ of the vertices $u$ in $V$.
   3.3 Repeat $k_1$ times:
       3.3.1 While there is any $\Delta > 0$ in the last $k_2$ iterations of this loop do:
             3.3.1.1 Let $v$ be the next vertex down the list $L$.
             3.3.1.2 Let $s \in S(\pi_l, v)$ maximizing $\Delta$ in 3.3.1.3.
             3.3.1.3 Let $\pi_l' = s(\pi_l)$, $\Delta = cost(\pi_l') - cost(\pi_l)$.
             3.3.1.4 If $\Delta > random(-p)$, set $\pi_l = \pi_l'$.
       3.3.2 Set $\beta_2$ according to current statistics.
       3.3.3 Perturb randomly the value of $\pi_l(u)$ for $\beta_2\%$ of the vertices $u$ in $V$.
   3.4 Update $\pi_p$ with the best $\pi_l$ visited in the last run of step 3.3.
4. Return $\pi_p$.
```

Figure 3.13: Parallel stochastic probe

All parameters in sequential version are also used in the parallel version. In addition, to make the search more effective, we introduce (or modify) some new parameters:

- *Perturbation Parameters*

  Two perturbation parameters β_1 and β_2 are introduced to escape some local optima. We use β_1 to provide an initial solution for the next searching stage, and β_2 to provide an initial solution for the next probe within a searching stage. β_1 is always greater than β_2 to ensure that the solution subspaces searched by different processors do not overlap, while each solution subspace is searched as thoroughly as possible. Figure 3.14 shows the effect of the various value of β_1/β_2 on the solution quality when we fix the other parameters for graph $R100\_2$ on 8 processors.

- *Loop Parameters*

  The parallel stochastic probe is divided into a sequence of searching phases,

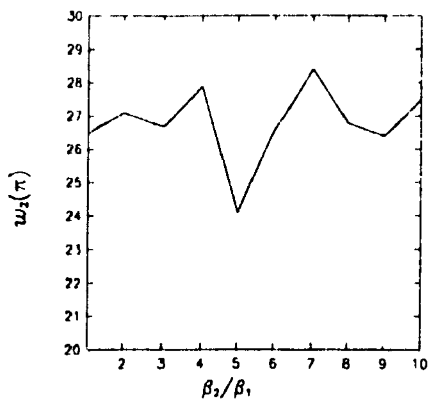

Figure 3.14: Tuning β_2/β_1 for graph $R100_2$

and each processor performs k_1 probes during each searching phase. The thoroughness of the solution subspace search is determined by k_2, and the number of probes performed during each searching phase is determined by k_1. A larger k_2 allows a more thorough probe search in the solution space with much more running time, and a larger k_1 increases the granularity of parallelization while decreases the number communications and coordinations among the processors.

Our experiments show that it is more appropriate to set k_1 at the range of $1 \sim 3$. Because sequential stochastic probe itself is composed of well-organized probes, it has some continuity. If k_1 is too large, the parallelism will destruct this configuration, and the possibility of missing a good solution will be increased.

3.5 Design of Experiments

In the following chapters, we present how we apply these heuristics to each of the five models we have designed for our workstation farms in Chapter 2 and what their performances are. To state them more concisely and clearly, we first discuss some issues related to our experiments in this section.

3.5.1 Experiment environment

All of the experiments of sequential heuristics are performed on a SUN Sparc 2 workstation running SUN-OS Release 4.01. All of the experiments of our parallel version of these heuristics are performed on a Butterfly GP1000.

3.5.2 Benchmark graph

The experiments can be classified into five parts, each part corresponding to one of the five designed models. Benchmark graphs are generated for each of them.

We generate our benchmark graphs based on the following considerations:

- They should represent the fundamental characteristics of the workstation farms' physical background;
- They can provide basis for repeatable experiments, and constitute a broad enough spectrum to yield insights into the general performance of our techniques when applied to the task assignment problem.

All the benchmark graphs in this thesis can be divided into two groups: one for uniform m-way graph partition model, one for the rest four models.

1. **uniform m-way graph partition model**

   We use two general classes of graphs for our performance comparisons: *random graphs* and *geometric graphs*. Both of the two classes of graphs are mainly characterized by two parameters: n, the vertex number, and d, the expected degree for each vertex.

   Random graph generation: Given n and d, define $p_r = d/(n-1)$. Value p_r specifies the probability that any given pair of vertices constitutes an edge. The vertex and edge weights are generated randomly in some specific integer ranges.

Geometric graph generation: Given n and d, define $k = \sqrt{d/(n\pi)}$. The coordinates of n vertices are first generated randomly on a unit square plane. Two vertices share a connecting edge if and only if the Euclidean distance between them is k or less. The vertex weights are generated randomly in a specific integer range. The weight for any edge is the ceiling integer of the product of a scale-factor S and the ratio of the distance between the vertices incident to the edge over k.

Our benchmark graphs are specified in Table 3.1.

Table 3.1: Characteristics of the Benchmark Graphs

name	n	d	w_1	w_2	S	d_{min}	d_{max}	$\lvert E\rvert$
*R*100_2	100	2	1–5	1–5		0	8	101
*G*100_2	100	2	1-5		10	0	6	91
*R*200_4	200	4	1–5	1-5		0	11	410
*G*200_4	200	4	1–5		10	0	8	387
*R*400_8	400	8	1–5	1-5		0	18	1625
*R*400_8	400	8	1–5		10	0	16	1471

The first letter of a graph name designates the graph class: R for random graph, and G for geometric graph. For each graph we specify its vertex number n, expected degree d, range for w_1, range for w_2 (for random graphs), scale factor S (for geometric graphs), minimum degree d_{min}, maximum degree d_{max}, and total edge number $|E|$. The last three entries are measured from the generated graph. We in general choose small d as most interesting applications involve graphs with a low average degree, and because such graphs are better for distinguishing the performance of different heuristics than denser ones [22].

2. **Nonuniform single-bus total cost model, Nonuniform single-bus completion cost model, General nonuniform total cost model**

The graphs used in these three models are organized into three categories, according to the communication pattern of a problem instance. The data set are basically generated following Lo's experiments designed in [28]. The communication pattern of a problem instance can be:

- *clustered*, in which there are roughly $3m$ clusters, and 40% of the communication costs in C are nonzero;
- *sparse*, in which only 1/6 of the elements in C are nonzero;
- *structured*, in which the inter-module communication pattern can be line, ring, square $2-D$ mesh, or binary tree.

Let n be the number of task modules, and m the number of processors. To model the situation in which some task modules cannot be executed on some processors, we set 5% of the execution costs in X to infinity.

For all of our data sets, the cost are randomly generated. All the computational cost, in $X(i,j)$ for $i \in [n]$, $j \in [n]$, range from 1 to 20. The amount of communication (number of messages) between any pair of communicating task modules, in matrix $Y(i,j)$ for $i,j \in [n]$, ranges from 1 to 10. All the interference costs, in matrix $I(k,i,j)$ for $i,j \in [n], k \in [m]$, range from 1 to 5. For clustered data sets, all inter-cluster costs range from 10 to 20. For our experiments on completely connected architectures, the communication cost matrix $C(i,j,u,v)$ was generated randomly (as opposed to the matrix $Y(i,j)$) with entries in the range 1 to 10.

3. General nonuniform completion cost model

Experiments for this model are classified into two parts based on the architecture interconnection network: (i) experiments based on the completely-connected network; (ii) experiments based on mesh and hypercube networks.

Graphs for the completely-connected networks are generated as those in last three models. For mesh and hypercube networks, we assume the program on these models has 256 tasks, and the system has 64 processors. We generate graphs in the same way as those for the completely-connected networks.

For total cost models (uniform or nonuniform), we generated graphs with n ranging from 60 $\sim$ 120 and m ranging from 6 $\sim$ 12, while for the completion cost models (uniform or nonuniform), we use smaller n and m with n ranging from 30 to 60 and m from 3 to 6. We choose smaller n and m for the completion cost models due to the fact that these two models are much more time consuming than the total cost models in experiments. So just for simplicity, we generate graphs with smaller n and m for the completion cost models, which are still good enough to represent our work without generality.

In the ensuring discussion and the tables, the name of each data set begins with "C" for clustered, "S" for sparse, or the name of topology for structures communication patterns, followed by n and m.

3.5.3 How we compare the performance?

The quality of a sequential algorithm is mainly determined by solution quality and execution time. The quality of a parallel algorithm is determined by, in addition to those for sequential algorithms, also the *speedup*[2] and the *processor utilization* or *efficiency*[3].

Experiments are conducted to compare the relative performances (solution quality and execution time) of sequential simulated annealing, tabu search and our stochastic probe heuristics, and to evaluate the speedup and solution quality of their parallel versions. For parallel experiments, we run every benchmark graph with 1 to 10 processors for all the five models. All our experimental results are presented in the following chapters model by model. To simplify presentation, we use SA, TA, and

[2]The running time of the parallel algorithm executed on one processor, divided by the running time of the parallel algorithm executed on a number of processors.

[3]The speedup divided by the number of processors used to execute the algorithm.

SP to denote our heuristics based on simulated annealing, tabu search and stochastic probe respectively. Each of our data sets contains one problem instance. All the values reported are averages over 10 runs of a heuristic, except for some time-consuming cases.

Chapter 4

Uniform M-way Graph Partition Model

This chapter considers the general single-bus homogeneous workstation farms, in which:

- All processors are homogeneous;
- All processors are connected by a high-speed bus;
- There are no competition for system resources between any pair of processes assigned on the same processor;
- Communication between any pair of processes within a processor is negligible.

As discussed in chapter 2, we can model this kind of workstation farm by m-way graph partition. We can model a parallel computation by an undirected graph $G = (V, E)$ in which each vertex in V represents a process and each edge in E represents a logical communication channel. The computation load of each process can be modeled by a function $w_1 : V \rightarrow I$ (I is the set of positive integers). The communication load of each channel can be modeled by a function $w_2 : E \rightarrow I$. Let $m > 0$ be the number of processors in a workstation farm. The task assignment problem for a workstation farm can thus be modeled by the following m-way graph partition problem: find a mapping $\pi : V \rightarrow \{1, 2, \ldots, m\}$ such that

$$W_2(\pi) = \sum_{\substack{e=\{u,v\}\in E \\ \pi(u)\neq\pi(v)}} w_2(e)$$

is minimized under the constraint that

$$W_1(\pi) = \sum_{1\leq i<j\leq m} |w_1(P_\pi(i)) - w_1(P_\pi(j))|$$

is minimal, where $P_\pi(i) = \{v \in V | \pi(v) = i\}$ for $1 \leq i \leq m$ (For any subset $C \subseteq V$,$w_1(C) = \sum_{v\in C} w_1(v)$).

In this chapter, we will apply all the above three techniques (both sequential algorithm and parallel algorithm) on this model. Experimental studies are presented in the following sections of this chapter to evaluate their performance on this uniform M-way graph partition Model for task assignment problem for all our benchmark graphs. Section 4.1 addresses some problem-specific design issuse. It is followed by our sequential experimental studies and parallel experimental studies in Section 4.2 and Section 4.3 respectively. Our observation of these techniques for the task assignment problem on this model is presented in Section 4.4.

4.1 Graph transformation

The time complexity of an iterative algorithm is largely determined by the efficiency by which the objective functions and the constraint conditions are evaluated. In this model, there are two objective functions involved, and they vary inconsistently. While $W_2(\pi)$ allows simple incremental update after each vertex move or vertex exchange operation, $W_1(\pi)$ needs at least $O(m)$ update steps after each of such operations. Therefore we adopt the following graph transformation in [26] to combine $W_1(\pi)$ and $W_2(\pi)$ into a single objective function easy for incremental evaluation.

Transformation algorithm:

Given a graph $G = (V, E)$ described in the last section we transform G into another graph $G^* = (V, E^*)$ where $E^* = \{\{u, v\}|u, v \in V\}$, and define a new edge

measurements	SP	SA	TS
W_1	2	2	2
W_2	22.5	28	39.6
CPU time	25.92	102.77	114.84

Table 4.1: Sequential performance comparison for R100_2 (m=3)

weight function $w_3 : E^* \rightarrow \Re$ ($\Re$ is the set of all positive real numbers) such that

$$w_3(e) = \begin{cases} w_1(u)w_1(v)R - w_2(e) & \text{if } e = \{u,v\} \in E; \\ w_1(u)w_1(v)R & \text{if } e = \{u,v\} \in E^* - E \end{cases}$$

where R is a positive real number called *augmenting factor.*

As pointed out in [26], if $R > \sum_{e \in E} w_2(e)$, any partition π that maximizes

$$W_3(\pi) = \sum_{\substack{e=\{u,v\}\in E^* \\ \pi(u)\neq\pi(v)}} w_3(e) = R \sum_{1 \leq i < j \leq m} w_1(P_\pi(i))w_1(P_\pi(j)) - W_2(\pi)$$

will minimize $W_2(\pi)$ under the constraint that $\sum_{1 \leq i < j \leq m} w_1(P_\pi(i))w_1(P_\pi(j))$ is maximized, which in turn is equivalent to minimizing $W_1(\pi)$ [41].

Based on the above transformation, from now on, we will focus on graph partitions π that maximize $W_3(\pi)$ in this chapter.

4.2 Experimental studies with sequential algorithms

Experiments are conducted to compare the relative performances of these three sequential techniques (simulated annealing, tabu search, and our stochastic probe) for all benchmark graphs (refer to Section 3.5.2) on this model. We run each algorithm 10 times for each graph, and report the details for the solution quality and execution time in Table 4.1 to Table 4.6.

We can conclude from these data that

1. For all our problem instances, our sequential SP always outperforms SA and TA in terms of solution quality. Compared with SA, it improves on the average

measurements	SP	SA	TS
W_1	2	2	2
W_2	0	0.4	0.9
CPU time	29.74	64.56	75.68

Table 4.2: Sequential performance comparison for G100_2 (m=3)

measurements	SP	SA	TS
W_1	0	0	0
W_2	286	340	331
CPU time	183.08	390.98	556.03

Table 4.3: Sequential performance comparison for R200_4 (m=5)

measurements	SP	SA	TS
W_1	0	0	0
W_2	12.8	34.1	29.0
CPU time	499.56	1021.92	1088.76

Table 4.4: Sequential performance comparison for G200_4 (m=5)

measurements	SP	SA	TS
W_1	24	24	24
W_2	2049	2165	2156
CPU time	891.02	1503.11	1491.97

Table 4.5: Sequential performance comparison for R400_8 (m=10)

measurements	SP	SA	TS
W_1	24	24	24
W_2	169	196	231
CPU time	598.16	22.47	2262.79

Table 4.6: Sequential performance comparison for G400_8 (m=10)

35.3% on the solution quality over all of our benchmark graphs; compared with TA, the average improvement is 40%.

2. For all our problem instances, our sequential SP always has minimal execution time. The average execution times for SA and TA are 2.37 and 2.93 times that for SP respectively.

3. For each of these problem instances, all the algorithms reach the same minimal $W_1(\pi)$.

Our sequential experiments demonstrate that SP always outperforms SA and TS both in solution quality and running time.

4.3 Experimental studies with parallel algorithms

Proc. No.	SP		SA		TA	
	W_2	speedup	W_2	speedup	W_2	speedup
1	22.5	1.00	24.0	1.00	34.0	1.00
2	22.6	1.97	30.5	1.74	36.2	1.95
3	22.5	2.90	31.5	2.33	34.5	2.65
4	22.2	3.83	29.0	2.76	33.5	3.44
5	22.6	4.74	31.0	3.09	32.2	4.11
6	22.2	5.62	27.3	3.44	36.7	4.66
7	23.8	6.61	31.0	3.74	32.8	5.01
8	23.8	7.46	30.8	3.90	34.2	5.36
9	23.5	8.25	32.5	3.93	32.5	5.65
10	23.6	9.05	28.0	4.16	32.3	5.76

Table 4.7: Speedup and W_2 for R100_2 (m=3), W_1=2

Parallel experiments are performed to evaluate the speedup nd solution quality for all the three techniques (simulated annealing, tabu search, and stochastic probe) in parallel version for the task assignment problem in this model. We run each of the

Proc. No.	SP		SA		TA	
	W_2	speedup	W_2	speedup	W_2	speedup
1	0.0	1.00	0.0	1.00	2.0	1.00
2	0.0	1.84	0.0	1.72	2,5	1.86
3	0.0	2.68	1.0	2.28	1.0	2.66
4	0.0	3.51	0.5	2.70	2.0	3.45
5	0.0	4.33	1.0	3.02	2.0	4.08
6	0.0	5.16	0.0	3.31	2.3	4.60
7	0.0	5.97	0.8	3.51	2.0	5.11
8	0.0	6.77	0.5	3.72	1.5	5.40
9	0.0	7.46	0.8	3.88	2.0	5.89
10	0.0	8,32	1.3	4.01	1.8	5.90

Table 4.8: Speedup and W_2 for G100_2 (m=3), W_1=0

three parallel algorithms on 1 to 10 processors for every one of the six benchmark graphs. The results for both the solution quality (W_1, W_2) and execution time are reported in Table 4.7 to Table 4.12.

4.3.1 Speedup evaluation

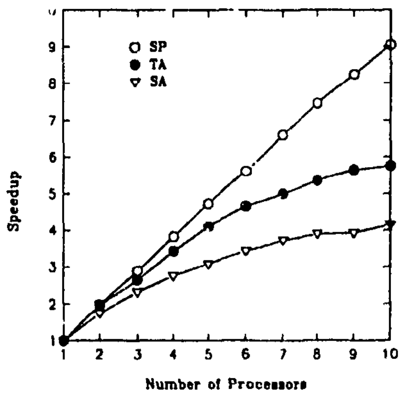

Figure 4.1: Speedup for graph R100_2 with up to 10 processors

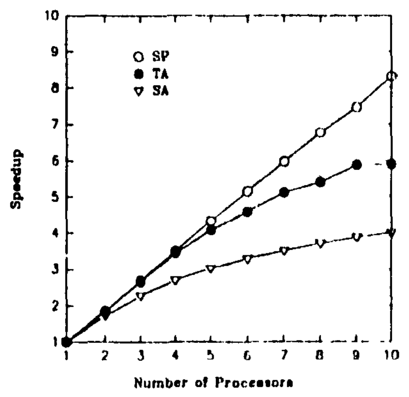

Figure 4.2: Speedup for graph G100_2 with up to 10 processors

Figure 4.1 to Figure 4.6 show the speedup obtained by the three parallel algorithms on each of our six benchmark graphs. From these figures, we can conclude that (with up to 10 processors) :

Proc. No.	SP		SA		TA	
	W_2	speedup	W_2	speedup	W_2	speedup
1	286.0	1.00	332.0	1.00	351.0	1.00
2	283.5	2.00	318.3	1.65	357.0	1.95
3	284.4	2.97	321.3	2.20	360.3	2.88
4	287.0	4.00	348.7	2.58	359.3	3.72
5	287.8	4.95	342.0	2.87	361.0	4.63
6	287.4	5.94	370.0	3.10	361.0	5.51
7	289.6	6.70	343.7	3.33	355.5	6.36
8	287.9	7.54	358.5	3.49	363.6	6.70
9	289.9	8.02	343.0	3.55	365.7	7.29
10	290.9	9.30	355.3	3.77	262.0	7.41

Table 4.9: Speedup and W_2 for R200_4 (m=5), W_1=0

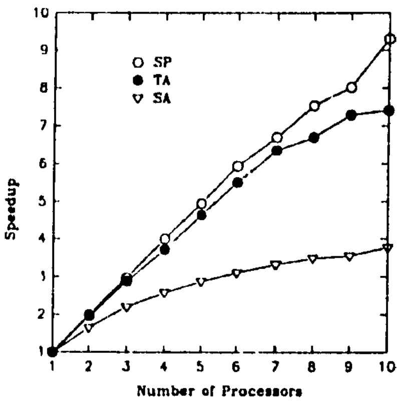

Figure 4.3: Speedup for graph R200_4 with up to 10 processors

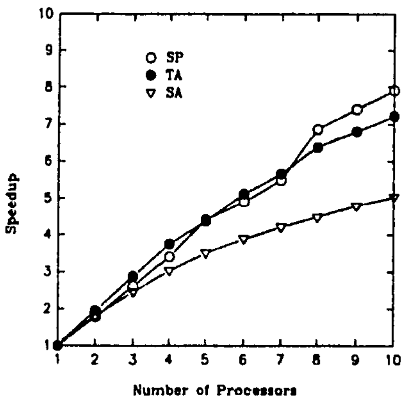

Figure 4.4: Speedup for graph G200_4 with up to 10 processors

1. Parallel stochastic probe can achieve almost linear speedup. With up to 10 processors, it obtains over 80% and 76% processor utilization for random graphs and geometric graphs respectively.

2. The speedup obtained in parallel tabu search varies with the problem size to some extent. The bigger the problem instance is, the better speedup obtained. With up to 10 processors, parallel tabu research can achieve over 60%, 70%, and 90% processor utilization for 100-node graphs, 200-node graphs, and 400-node graphs respectively.

Proc. No.	SP		SA		TA	
	W_2	speedup	W_2	speedup	W_2	speedup
1	12.8	1.00	44.0	1.00	47.0	1.00
2	13.1	1.79	33.0	1.81	50.5	1.95
3	14.4	2.62	34.2	2.46	52.4	2.88
4	14.2	3.41	41.8	3.03	48.7	3.75
5	14.1	4.41	42.0	3.50	49.8	4.36
6	14.9	4.90	42.3	3.89	45.8	5.11
7	13.7	5.49	54.0	4.21	50.7	5.66
8	13.1	6.86	51.7	4.48	45.5	6.38
9	14.5	7.40	49.0	4.79	41.7	6.80
10	14.0	7.92	50.7	5.03	51.5	7.22

Table 4.10: Speedup and W_2 for G200_4 (m=5), W_1=0

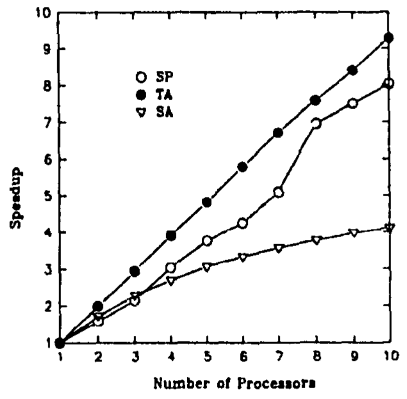

Figure 4.5: Speedup for graph *R*400_8 with up to 10 processors

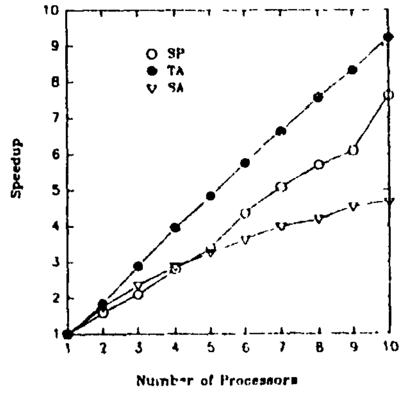

Figure 4.6: Speedup for graph *G*400_8 with up to 10 processors

3. Parallel simulated annealing achieves generally increasing speedup with using up to 10 processors. And it performs a little better with geometric graphs than with random graphs. Around 30% and 40% speedup are obtained in random graphs and geometric graphs respectively.

4. For all problem instances, parallel stochastic probe can obtain the best average speedup.

Proc. No.	SP		SA		TA	
	W_2	speedup	W_2	speedup	W_2	speedup
1	2049	1.00	2118	1.00	2125	1.00
2	2043	1.59	2146	1.72	2121	1.99
3	2045	2.15	2124	2.29	2121	2.94
4	2057	3.04	2188	2.70	2109	3.91
5	2060	3.77	2192	3.06	2122	4.82
6	2063	4.26	2246	3.34	2148	5.79
7	2065	5.09	2221	3.58	2126	6.71
8	2092	6.96	2220	3.78	2157	7.59
9	2070	7.51	2248	3.98	2125	8.41
10	2076	8.04	2270	4.12	2120	9,29

Table 4.11: Speedup and W_2 for R400_8 (m=10), W_1=24

4.3.2 Solution quality comparison

From Table 4.7 to Table 4.12, we can draw the following conclusions on the solution quality:

1. For all the problem instances, parallel stochastic probe can always outperform the parallel simulated annealing and tabu search in terms of solution quality. Table 4.13 represents the average W_2 obtained from the three parallel algorithms over the six benchmaık graphs.

2. For all problem instances, the solution obtained from the parallel versions of tabu search, simulated annealing and our stochastic probe are comparable to those of their corresponding sequential versions.

3. For parallel SP, SA, and TA, the average fluctuations for W_2 over different processors for all benchmark graphs are 5.87%, 20.4% and 47.4% respectively.

4.4 Observation

From the above experiments, we can make the following observations:

Proc. No.	SP		SA		TA	
	W_2	speedup	W_2	speedup	W_2	speedup
1	169.7	1.00	161.0	1.00	198.0	1.00
2	170.6	1.60	155.5	1.75	232.3	1.84
3	167.0	2.10	184.0	2.36	223.0	2.89
4	170.4	2.80	188.5	2.88	221.8	3.95
5	169.2	3.40	218.5	3.26	223.0	4.83
6	173.0	4.36	218.0	3.63	226.2	5.76
7	167.1	5.08	223.5	3.99	223.8	6.63
8	176.9	5.70	245.5	4.20	212.7	7.58
9	171.5	6.09	245.5	4.54	224.8	8.32
10	182.7	7.62	278.5	4.67	221.4	9.21

Table 4.12: Speedup and W_2 for G400_8 (m=10), W_1=24

Graph	W_2		
	SP	SA	TA
R100-2	22.7	29.6	33.4
G100-2	0	0.5	1.9
R200-4	287.4	343.3	359.6
G200-4	13.9	44.3	48.4
R400-8	2062.2	2197.2	2127.4
G400-8	171.8	211.9	220.7

Table 4.13: Average solution cost (W_2) over all experiments for each benchmark graph with the three parallel algorithms

1. Our sequential stochastic probe always yields the best solutions for all the problem instances with less CPU time.

2. Parallel stochastic probe obtains almost linear speedup by using up to 10 processors with solution quality comparable to those of the sequential version; parallel tabu search obtains over 60% processor utilization with up to 10 processors with solution quality comparable to those of its sequential version; parallel simulated annealing obtains over 30% processor utilization with up to 10 processors with the solution quality comparable to those of its sequential version.

3. The speedup in parallel tabu search varies with the problem size. The bigger the problem is, the better speedup the parallel algorithm gets. This is due to the way we parallelize it (parallelizing the aggressive search at each iteration). To speak more concisely, the speedup depends on the the percentage of program parallelized (the aggressive search part) as we describe in Figure 3.5.

4. Parallel simulated annealing achieves stable speedup over all the problem instances. With 10 processors, it achieves over 30% processors utilization, which is not worse than those reported from Witte E.E's speculative parallel simulated annealing (refer to Section 3.2.2).

Chapter 5

Nonuniform Single-bus Total Cost Model

The workstation farm considered in this chapter is a general heterogeneous system connected by a high-speed single bus. In particular, the system has the following characteristics:

- All processors can be heterogeneous;
- All processors are connected by a high-speed bus;
- Communication between any pair of processes within a processor is negligible.

The system performance is mainly affected by three of costs: *execution cost*, *communication cost*, and *interference cost*, among which the *communication cost* and *interference cost* are processor independent.

As stated in chapter 2, the task assignment on this kind of workstation farms can be modeled as the Nonuniform Single-bus Total Cost Model. The task assignment problem is thus to find a mapping $\pi : [n] \rightarrow [m]$ to minimize the *total cost*

$$cost(\pi) = \sum_{i=1}^{n} X(i, \pi(i))) + \sum_{\substack{\pi(i) \neq \pi(j) \\ i<j}} C(i,j) + \sum_{\substack{\pi(i) = \pi(j) \\ i<j}} I(i,j)$$

.

In this chapter, we apply all the above three techniques (both sequential algorithm and parallel algorithm) to solve the task assignment problem on the model. Experimental studies are presented in the following sections of this chapter to evaluate their performances on this Nonuniform Single-bus Total Cost Model for task assignment problem for all our benchmark graphs. Section 5.1 and 5.2 present our sequential experimental studies and parallel experimental studies respectively, followed by our observation of these techniques for the task assignment problem on this model in Section 5.3.

5.1 Experimental studies of sequential algorithms

	C60-6			C80-8			C100-10			C120-12		
	SP	SA	TA	SP	SA	TA	SP	SA	TA	SP	SA	TA
cost	5152	5152	5152	8622	8622	8622	14803	15362	15257	20571	20571	20571
CPU (sec.)	0.1	1.6	0.3	0.3	0.6	0.3	0.5	1.3	0.6	0.9	1.9	0.9

Table 5.1: Performance comparisons of sequential algorithms for clustered data sets

	S60-6			S80-8			S100-10			S120-12		
	SP	SA	TA	SP	SA	TA	SP	SA	TA	SP	SA	TA
cost	1805	1817	1818	3054	3065	3089	4628	4657	4645	6469	6509	6503
CPU (sec.)	0.2	1.8	1.8	0.8	6.1	4.7	1.2	3.5	3.3	1.5	8.3	4.3

Table 5.2: Performance comparison of sequential algorithms for sparse data sets

We conduct sequential experiments to compare the performances of the three algorithms in sequential version (simulated annealing, tabu search, and our stochastic probe) for the task assignment on this *uniform total cost model*. We run each algorithm 10 times for each benchmark graph (Section 3.5.3). The experimental results

	Line60-6			Ring60-6			Mesh49-6			Tree60-6		
	SP	SA	TA	SP	SA	TA	SP	SA	TA	SP	SA	TA
cost	996	997	1013	1697	1709	1704	817	821	822	1002	1009	1011
CPU (sec.)	0.6	6.1	2.6	0.5	1.7	2.0	0.1	1.2	0.7	0.5	3.5	2.6

Table 5.3: Performance comparison of sequential algorithms for structured data sets

for clustered, sparse, and structured data sets are reported in Table 5.1 to Table 5.3. From these results, we can draw the following conclusions:

1. Compared with SA and TA, the average improvements of SP for solution quality (*cost*) over the 12 problem instances are 0.67% and 0.79% respectively.
2. In terms of computation time, the average CPU times for SA and TA are 6.69 and 3.94 times of those for SP respectively.

Our sequential experiments demonstrate that, for the task assignment on this model, SP always outperforms SA and TA both in solution quality and running time.

5.2 Experimental studies with parallel algorithms

To evaluate the performance of parallel simulated annealing, parallel tabu search, and parallel stochastic probe for the task assignment on this model, we run each of these three parallel algorithms on 1 to 10 processors for all our benchmark graphs generated for this model. The experimental results of both solution quality and speedup are reported in Table 5.4 to Table 5.15.

5.2.1 Speedup evaluation

Figure 5.1 to Figure 5.12 show the speedup obtained by each of the three parallel algorithms in the 12 problem instances for different types of data sets (clustered,

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	5220	1.00	5165	1.00	5152	1.00
2	5220	1.94	5165	1.78	5152	1.50
3	5220	2.82	5165	2.41	5152	1.68
4	5220	3.65	5165	2.89	5152	1.87
5	5220	4.43	5165	3.32	5152	1.90
6	5220	5.15	5165	3.68	5152	1.96
7	5220	5.86	5165	3.98	5152	2.18
8	5220	6.52	5165	4.21	5152	1.99
9	5220	7.08	5165	4.42	5152	2.16
10	5220	7.68	5165	4.62	5152	1.97

Table 5.4: Cost and speedup for Graph $C60\_6$

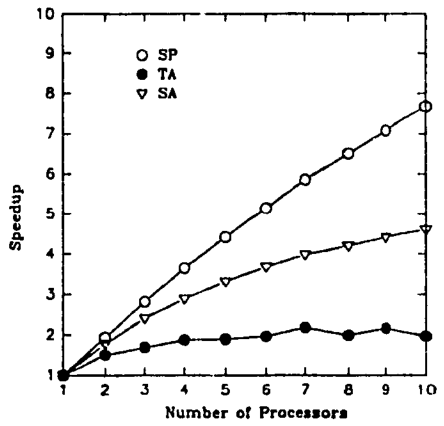

Figure 5.1: Speedup for graph $C60\_6$ with up to 10 processors

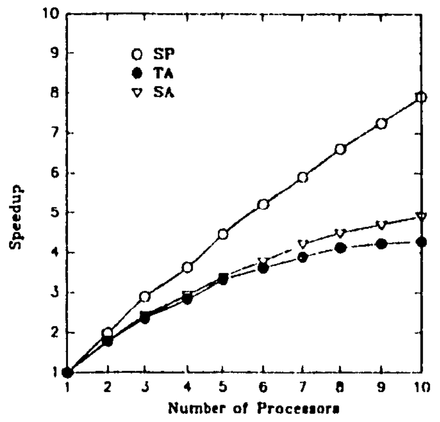

Figure 5.2: Speedup for graph $S60\_6$ with up to 10 processors

sparse, and structured). From these figures, we can conclude that (with up 1 to 10 processors):

1. Parallel stochastic probe can achieve almost linear speedup. With up to 10 processors, it obtains around 80% processor utilization for all the problem instances.

2. Both parallel tabu search and simulated annealing achieve moderate speedup. With 10 processors, around 35% processor utilization is obtained. However, the

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	1809	1.00	1803	1.00	1818	1.00
2	1811	1.98	1813	1.79	1818	1.78
3	1811	2.88	1809	2.42	1818	2.35
4	1809	3.63	1820	2.94	1818	2.83
5	1812	4.45	1816	3.38	1818	3.31
6	1811	5.22	1826	3.78	1825	3.61
7	1810	5.91	1827	4.22	1818	3.90
8	1812	6.61	1823	4.49	1818	4.12
9	1810	7.25	1825	4.70	1818	4.23
10	1810	7.91	1824	4.90	1825	4.27

Table 5.5: Cost and Speedup for Graph $S60\_6$

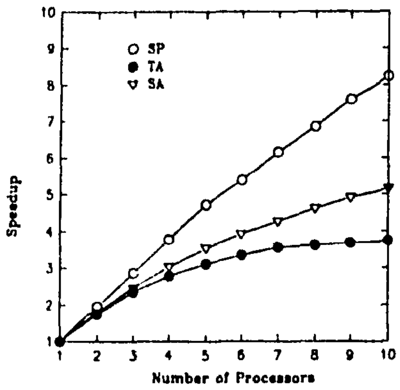

Figure 5.3: Speedup for graph $C80\_8$ with up to 10 processors

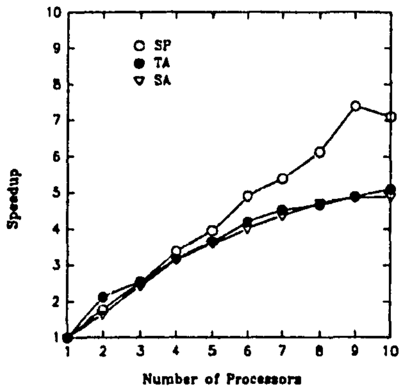

Figure 5.4: Speedup for graph $S80\_8$ with up to 10 processors

speedup of parallel tabu search varies a little around the speedup of parallel simulated annealing. As an extreme, for Graph $C60\_6$, it only gets speedup less than 2. This is due to the fact that the parallel rate (defined in Section 3.2) in this problem is very low.

3. For all the problem instances, parallel stochastic probe can obtain best speedup among the three parallel algorithms.

4. Processor utilization decreases with more processors used.

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	8850	1.00	8856	1.00	8622	1.00
2	8695	1.96	8622	1.80	8622	1.74
3	8695	2.87	8622	2.47	8622	2.36
4	8695	3.78	8622	3.03	8622	2.78
5	8695	4.73	8622	3.54	8622	3.12
6	8695	5.39	8622	3.93	8622	3.35
7	8695	6.16	8622	4.27	8622	3.56
8	8695	6.86	8622	4.62	8622	3.62
9	8695	7.60	8622	4.92	8622	3.69
10	8695	8.23	8622	5.16	8622	3.74

Table 5.6: Cost and speedup for Graph $C80\_8$

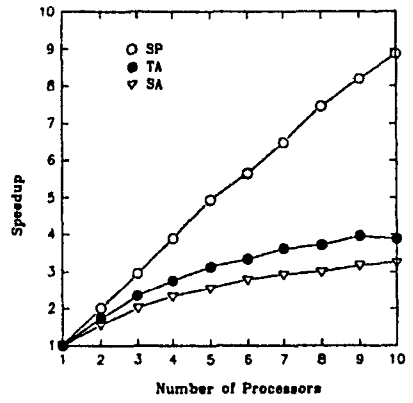

Figure 5.5: Speedup for graph $C100\_10$ with up to 10 processors

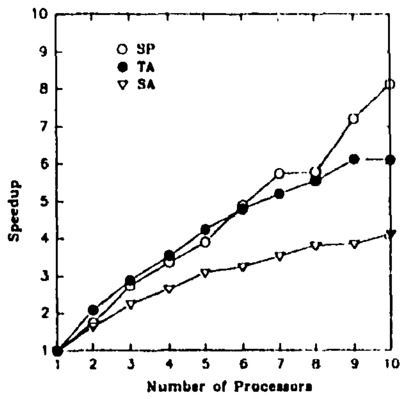

Figure 5.6: Speedup for graph $S100\_10$ with up to 10 processors

5.2.2 Solution quality comparison

Table 5.16 summaries Table 5.4 to Table 5.15 on solution quality by showing the average cost of each algorithm for each problem instance over different number of processors. From this table, we can draw the following conclusions in terms of solution quality:

1. For the 8 sparse and structured problem instances, parallel SP improves the average solution quality of parallel SA and TA by 0.52% and 0.69% respectively. For the other 4 clustered problem instances, parallel SP gets solutions with a

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	3032	1.00	3046	1.00	3060	1.00
2	3034	1.78	3051	1.64	3046	2.12
3	3034	2.50	3053	2.43	3054	2.54
4	3034	3.37	3062	3.14	3054	3.16
5	3041	3.95	3064	3.59	3056	3.64
6	3051	4.90	3054	4.01	3059	4.19
7	3041	5.38	3061	4.36	3047	4.51
8	3042	6.13	3054	4.72	3049	4.66
9	3044	7.39	3072	4.86	3060	4.88
10	3044	7.10	3048	4.89	3079	5.10

Table 5.7: Cost and Speedup for Graph $S80\_8$

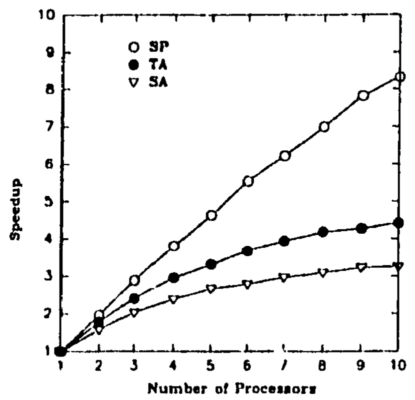

Figure 5.7: Speedup for graph $C120\_12$ with up to 10 processors

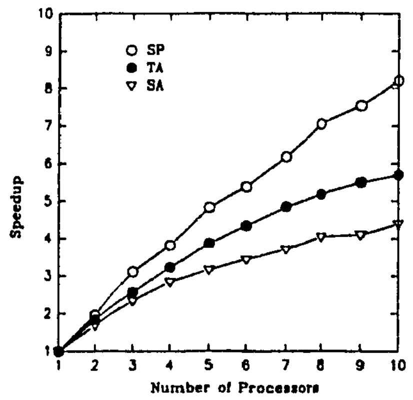

Figure 5.8: Speedup for graph $S120\_12$ with up to 10 processors

little worse solution quality than those of parallel SA and TA.

2. For all problem instances, the solutions obtained from the parallel versions of tabu search, simulated annealing and our stochastic probe are comparable to those of their corresponding sequential versions.

3. For parallel SP, SA, and TA, the average fluctuations for *cost* over different processors for all benchmark graphs are 0.52%, 1.38% and 1.04% respectively.

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	15229	1.00	15220	1.00	15257	1.00
2	15155	2.01	14830	1.57	15257	1.75
3	15189	2.96	14830	2.02	15257	2.35
4	15190	3.90	14830	2.34	15257	2.76
5	15155	4.94	14830	2.56	15257	3.14
6	15190	5.65	14830	2.79	15257	3.34
7	15155	6.49	14830	2.93	15257	3.62
8	15154	7.46	14830	3.01	15257	3.72
9	15155	8.20	14830	3.18	15257	3.96
10	15155	8.88	14830	3.26	15257	3.88

Table 5.8: Cost and speedup for Graph C100_10

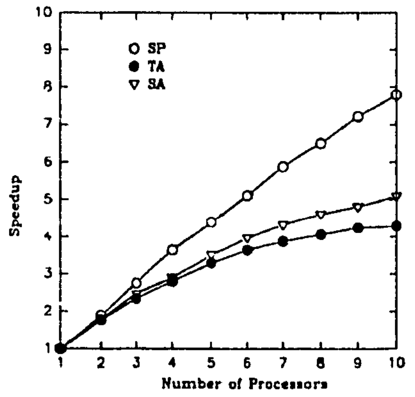

Figure 5.9: Speedup for graph *Line*60_6 with up to 10 processors

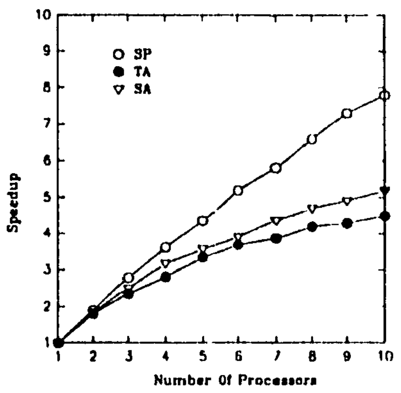

Figure 5.10: Speedup for graph *Ring*60_6 with up to 10 processors

5.3 Observation

From the above experiments, we can make the following observations:

1. Our sequential stochastic probe always yields the best solution quality for the all problem instances with less CPU time on this *nonuniform single-bus total cost model.*

2. Parallel stochastic probe obtains almost linear speedup by using up to 10 processors with solution quality comparable to those of the sequential versions;

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	4607	1.00	4624	1.00	4630	1.00
2	4613	1.74	4628	1.64	4658	2.07
3	4626	2.73	4619	2.24	4623	2.88
4	4622	3.36	4646	2.65	4624	3.53
5	4620	3.90	4648	3.09	4650	4.23
6	4618	4.89	4639	3.25	4638	4.79
7	4634	5.74	4626	3.53	4636	5.20
8	4617	5.78	4654	3.81	4628	5.54
9	4624	7.21	4677	3.85	4628	6.12
10	4621	8.12	4660	4.12	4647	6.11

Table 5.9: Cost and speedup for Graph $S100\_10$

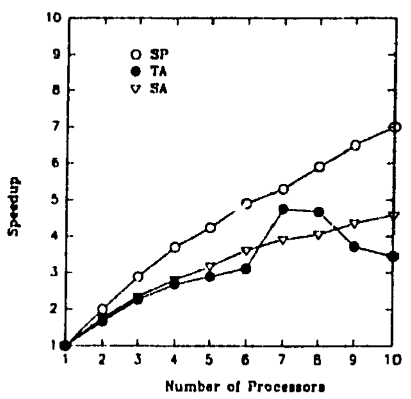

Figure 5.11: Speedup for graph $Mesh49\_6$ with up to 10 processors

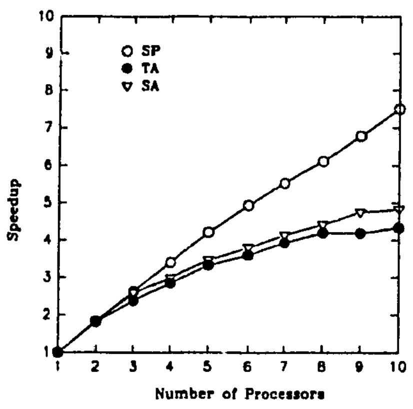

Figure 5.12: Speedup for graph $Tree60\_6$ with up to 10 processors

parallel tabu and simulated annealing achieves moderate speedup with up to 10 processors, and 45.2% and 43.1% average processor utilizations are obtained respectively.

3. The speedup on parallel tabu search varies with the problem size: the bigger the problem instance is, the better speedup the parallel algorithm gets. This is due to the way we parallelize it (parallelizing the aggressive search at each iteration).

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	20915	1.00	20910	1.00	20571	1.00
2	20915	1.97	20684	1.58	20571	1.79
3	20915	2.88	20684	2.03	20571	2.40
4	20914	3.81	20684	2.39	20571	2.95
5	20914	4.64	20684	2.67	20571	3.32
6	20914	5.54	20629	2.78	20571	3.67
7	20913	6.22	20629	2.97	20571	3.94
8	20914	7.00	20749	3.10	20571	4.20
9	20915	7.82	20749	3.23	20571	4.28
10	20914	8.32	20571	3.25	20571	4.42

Table 5.10: Cost and speedup for Graph $C120\_12$

4. The speedup achieved by this parallel simulated annealing for our task assignment on this model is better than that obtained by Witte's speculative parallel simulated annealing.

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	6432	1.00	6486	1.00	6569	1.00
2	6435	1.96	6504	1.69	6546	1.84
3	6451	3.11	6466	2.34	6522	2.57
4	6446	3.82	6470	2.85	6546	3.23
5	6449	4.83	6496	3.16	6534	3.87
6	6446	5.38	6475	3.44	6473	4.34
7	6450	6.17	6495	3.72	6405	4.85
8	6449	7.04	6486	4.05	6528	5.17
9	7650	7.54	6481	4.11	6489	5.50
10	6464	8.20	6487	4.40	6501	5.70

Table 5.11: Cost and speedup for Graph $S120\_12$

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	1698	1.00	998	1.00	1013	1.00
2	1698	1.89	998	1.81	1018	1.78
3	1698	2.75	1003	2.46	1018	2.34
4	1700	3.64	1009	2.90	1017	2.80
5	1700	4.39	1000	3.51	1018	3.30
6	1700	5.11	1006	3.97	1013	3.64
7	1699	5.89	1013	4.32	1013	3.88
8	1699	6.50	1017	4.61	1018	4.06
9	1701	7.22	1009	4.81	1013	4.24
10	1701	7.80	1012	5.09	1018	4.28

Table 5.12: Cost and Speedup fro Graph $Line60\_6c$

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	1698	1.00	1712	1.00	1715	1.00
2	1698	1.89	1709	1.83	1697	1.80
3	1700	2.78	1710	2.49	1697	2.34
4	1700	3.62	1720	3.18	1714	2.80
5	1699	4.35	1713	3.57	1697	3.34
6	1700	5.19	1716	3.90	1728	3.68
7	1698	5.81	1714	4.36	1725	3.87
8	1703	6.59	1708	4.70	1697	4.18
9	1700	7.30	1711	4.91	1697	4.28
10	1698	7.80	1718	5.19	1697	4.49

Table 5.13: Cost and speedup for Graph *Ring*60_6

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	816	1.00	825	1.00	816	1.00
2	820	1.99	826	1.76	847	1.67
3	824	2.88	816	2.34	821	2.26
4	822	3.69	827	2.81	822	2.69
5	820	4.24	818	3.17	821	2.87
6	821	4.90	816	3.61	816	3.12
7	818	5.30	820	3.91	835	4.75
8	819	5.91	821	4.06	839	4.68
9	821	6.50	824	4.36	828	3.72
10	819	7.00	831	4.58	821	3.44

Table 5.14: Cost and speedup for Graph *Mesh*49_6

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	1003	1.00	1006	1.00	1002	1.00
2	1005	1.82	1005	1.83	1009	1.81
3	1005	2.61	1009	2.58	1009	2.38
4	1005	3.39	1010	2.98	1015	2.85
5	1007	4.21	1011	3.46	1019	3.32
6	1008	4.94	1014	3.80	1015	3.60
7	1007	5.53	1014	4.13	1009	3.93
8	1004	6.11	1015	4.42	1009	4.20
9	1007	6.79	1016	4.75	1009	4.19
10	1008	7.51	1017	4.83	1009	4.34

Table 5.15: Cost and speedup for Graph *Tree*60_6

Graph	Average Cost		
	SP	SA	TA
C60-6	5220.0	5156.0	5152.0
S60-6	1810.5	1818.6	1819.4
C80-8	8710.5	8645.4	8622.0
S80-8	3039.7	3045.5	3056.4
C100-10	15172.7	14869.0	15257.0
S100-10	4266.2	4642.1	4636.2
C120-12	20914.3	20697.3	2057.1
S120-12	6447.2	6484.6	6511.2
Line60-6	1001.4	1006.5	1015.9
Ring60-6	1699.4	1713.1	1706.4
Mesh60-6	820.0	822.4	826.6
Tree60-6	1005.9	1011.7	1010.5

Table 5.16: Average solution cost over all experiments for each benchmark graph with the three parallel algorithms

Chapter 6

Nonuniform Single-bus Completion Cost Model

In this chapter, we study the Nonuniform Single-bus Completion Cost Model which is proposed for a general heterogeneous high-speed single bus workstation farm. In this workstation farm,

- All processors can be heterogeneous;
- All processors are connected by a high-speed bus;
- Communication between any pair of processes within a processor is negligible;
- The system objective is to minimize program completion cost.

Similar to the system in Chapter 5, the *execution cost*, *communication cost*, and *interference cost* are three key factors for the performance of the system, among which the *communication cost* and *interference cost* are processor independent.

Thus the task assignment on this kind of workstation farms becomes a problem to find a mapping $\pi : [n] \rightarrow [m]$ to minimize the *completion cost*

$$cost(\pi) = \max_{1 \le k \le m} \{ \sum_{i=1}^{n} X(i, \pi(i))) + \sum_{\substack{\pi(i) \neq \pi(j) \\ i < j}} C(i,j) + \sum_{\substack{\pi(i) = \pi(j) \\ i < j}} I(i,j) \}$$

.

In the following sections, we present our experimental studies on this model. We apply all the above three techniques (both sequential algorithm and parallel algorithm) to the task assignment problem on the model. Section 6.1 and Section 6.2 present our sequential experimental studies and parallel experimental studies respectively, and in Section 6.3 we make some observations on these experiments.

6.1 Experimental studies with sequential algorithms

	C30-3			C40-4			C50-5			C60-6		
	SP	SA	TA	SP	SA	TA	SP	SA	TA	SP	SA	TA
cost	976.3	976.8	994.5	1461.4	1501.7	1472.8	1965.3	1977.6	1976.4	2587.3	2590.7	2590.6
CPU (sec.)	0.2	3.3	6.7	1.6	8.8	4.1	6.1	14.8	42.4	7.0	8.7	10.3

Table 6.1: Performance comparison for clustered data sets

	S30-3			S40-4			S50-5			S60-6		
	SP	SA	TA	SP	SA	TA	SP	SA	TA	SP	SA	TA
cost	248.4	252.7	254.5	277.3	280.7	284.4	350.4	350.8	352.7	461.2	473.4	464.8
CPU (sec.)	0.2	1.3	2.6	1.1	1.4	4.9	1.7	6.5	8.9	6.2	8.3	15.4

Table 6.2: Performance comparison fro sparse data sets

	Line60-6			Ring60-6			Mesh49-6			Tree60-6		
	SP	SA	TA	SP	SA	TA	SP	SA	TA	SP	SA	TA
cost	200.3	202.6	208.4	435.5	437.4	437.8	180.4	180.7	194.5	196.3	199.4	234.2
CPU (sec.)	1.6	11.2	18.3	8.1	9.1	28.9	5.0	9.5	17.2	2.7	9.3	10.0

Table 6.3: Performance comparison for structured data sets

We conduct sequential experiments to compare the performance of the three algorithms in sequential version (simulated annealing, tabu search, and our stochastic probe) for the task assignment on this *nonuniform single-bus completion cost model*. We run each algorithm 10 times for each benchmark graph (Section 3.5.3). The experimental results for clustered, sparse, and structured data sets are reported in Table 6.1 to Table 6.3. From these results, we can draw the following conclusions:

1. Compared with SA and TA, the average improvements of SP for solution quality (*cost*) over the 12 problem instances are 1.0% and 2.5% respectively.

2. In terms of computation time, the average CPU times for SA and TA are 14.8 and 4.42 times of those for SP respectively.

Our sequential experiments show that, for the task assignment on this model, SP always outperforms SA and TA in both solution quality and running time.

6.2 Experimental studies with parallel algorithms

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	976	1.00	992	1.00	1003	1.00
2	981	1.93	998	1.99	998	1.90
3	984	2.86	999	2.98	989	2.65
4	985	3.66	976	3.95	985	2.98
5	983	4.58	976	4.90	989	3.59
6	985	5.38	976	5.87	991	4.40
7	983	6.12	976	6.81	979	4.30
8	995	7.12	976	7.72	985	4.45
9	985	7.60	976	8.58	988	5.23
10	986	8.56	976	9.50	998	5.17

Table 6.4: Cost and Speedup for Graph C30_3

To evaluate the performance of parallel simulated annealing, parallel tabu search, and parallel stochastic probe for the task assignment on this model, we run each of

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	249	1.00	261	1.00	260	1.00
2	253	1.91	260	1.97	253	1.87
3	253	2.81	260	2.90	260	2.64
4	254	3.78	261	3.80	259	2.99
5	251	4.53	258	4.64	258	3.66
6	254	5.25	261	5.38	258	4.57
7	254	6.30	251	6.27	256	4.47
8	255	6.73	257	6.91	259	4.85
9	254	8.11	262	7.51	251	5.66
10	254	8.17	262	8.12	254	5.83

Table 6.5: Cost and speedup for Graph S30_3

these three parallel algorithms on 1 to 10 processors for all our benchmark graphs generated for this model. The experimental results for both solution quality and speedup are reported in Table 6.4 to Table 6.15.

6.2.1 Speedup evaluation

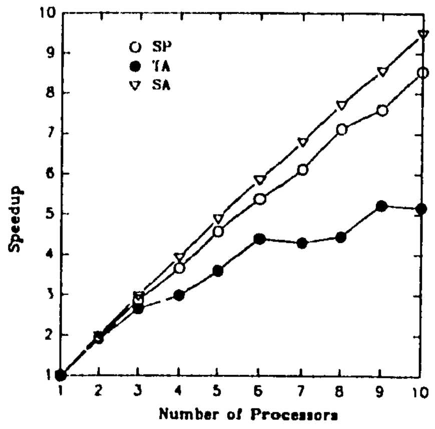

Figure 6.1: Speedup for graph $C30\_3$ with up to 10 processors

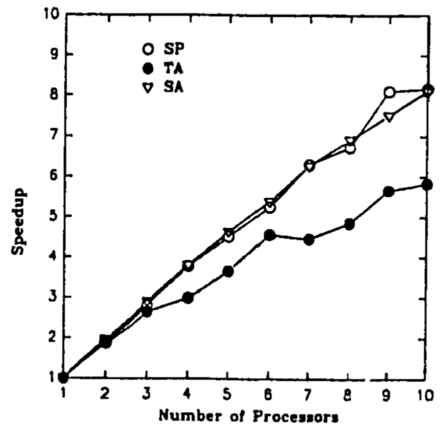

Figure 6.2: Speedup for graph $S30\_3$ with up to 10 processors

Figure 6.1 to Figure 6.12 show the speedup obtained by each of the three parallel algorithms in the 12 problem instances for different types of data sets (clustered, sparse, and structured). From these figures, we can conclude that (with up 1 to 10

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	1474	1.00	1470	1.00	1472	1.00
2	1475	1.94	1489	2.00	1507	1.91
3	1469	2.84	1487	2.99	1493	2.65
4	1476	3.72	1476	3.97	1470	3.43
5	1480	4.56	1468	4.95	1474	4.01
6	1490	5.47	1469	5.92	1499	4.45
7	1477	6.32	1474	6.90	1486	5.06
8	1477	6.86	1474	7.86	1462	5.89
9	1486	7.51	1477	8.79	1491	5.46
10	1477	8.61	1469	9.73	1506	5.74

Table 6.6: Cost and speedup for Graph C40_4

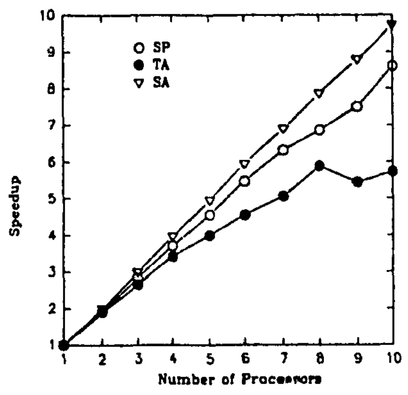

Figure 6.3: Speedup for graph $C40\_4$ with up to 10 processors

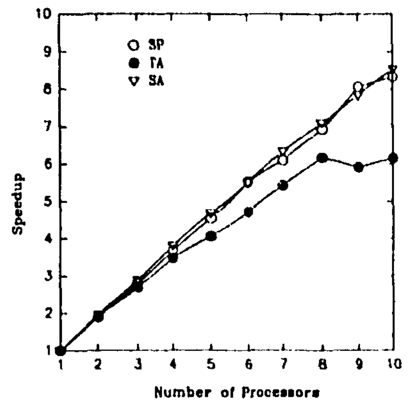

Figure 6.4: Speedup for graph $S40\_4$ with up to 10 processors

processors):

1. Both parallel SP and SA achieve almost linear speedup for all problem instances. Processor utilization decreases gradually. With 10 processors, 8.88 and 9.03 average speedup are obtained by parallel SP and SA.

2. Parallel TA achieves almost linear speedup for most of problem instances, except two 30-node graphs. This is due to that the parallelization rate here determined by aggressive search is much lower than those in other problem instances. With 10 processors, it obtains 6.84 average speedup over all the 12

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	284	1.00	283	1.00	283	1.00
2	279	1.91	277	1.97	279	1.92
3	283	2.80	291	2.90	281	2.70
4	280	3.68	298	3.81	280	3.48
5	284	4.54	282	4.69	282	4.07
6	283	5.53	288	5.52	280	4.72
7	282	6.10	293	6.34	285	5.43
8	280	6.94	286	7.11	282	6.17
9	286	8.06	286	7.85	280	5.91
10	284	8.33	289	8.54	282	6.16

Table 6.7: Cost and speedup for Graph S40_4

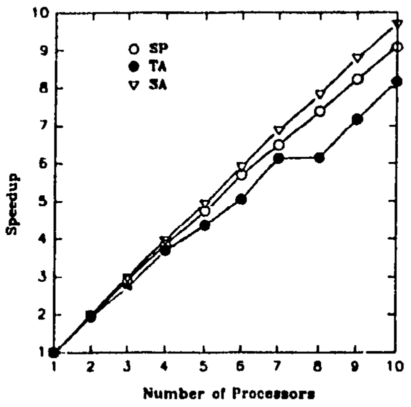

Figure 6.5: Speedup for graph $C50\_5$ with up to 10 processors

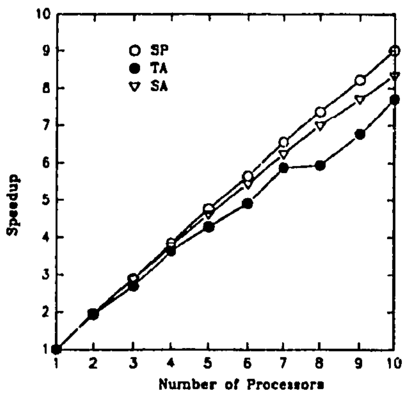

Figure 6.6: Speedup for graph $S50\_5$ with up to 10 processors

problem instances. Processor utilization decreases when more processors are used.

6.2.2 Solution quality comparison

Table 6.16 summaries Table 6.4 to Table 6.15 on solution quality by showing the average cost of each algorithm for each problem instance over different number of processors. From this table, we can draw the following conclusions in terms of solution quality:

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	1998	1.00	1979	1.00	2006	1.00
2	1984	1.96	1996	2.00	1974	1.94
3	1990	2.90	1996	2.99	1993	2.71
4	1986	3.84	1998	3.97	1983	3.69
5	1996	4.74	1996	4.95	2000	4.36
6	1987	5.69	2006	5.91	1999	5.05
7	1979	6.48	1998	6.88	1974	6.13
8	1993	7.39	1999	7.84	1987	6.15
9	1993	8.23	1991	8.80	1991	7.18
10	1989	9.09	1993	9.71	1975	8.17

Table 6.8: Cost and speedup for Graph C50_5

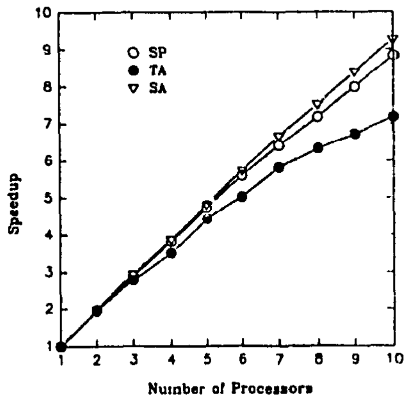

Figure 6.7: Speedup for graph $C60\_6$ with up to 10 processors

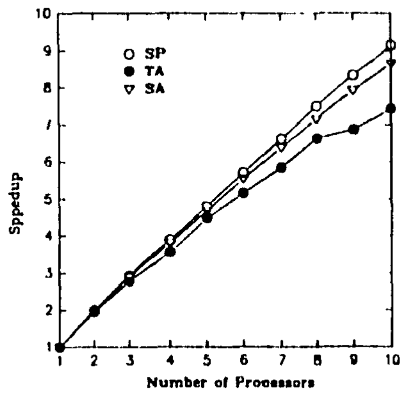

Figure 6.8: Speedup for graph $S60\_6$ with up to 10 processors

1. For all the 12 problem instances, parallel SP improves the average solution quality of parallel SA and TA by 2.26% and 1.79% respectively.

2. For all problem instances, the solution obtained from the parallel versions of tabu search, simulated annealing and our stochastic probe are comparable to those of their corresponding sequential versions.

3. For parallel SP, SA, and TA, the average fluctuations for *cost* over different processors for all benchmark graphs are 2.37%, 3.70% and 6.02% respectively.

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	348	1.00	346	1.00	358	1.00
2	252	1.96	348	1.96	386	1.93
3	350	2.90	354	2.88	350	2.70
4	346	3.84	355	3.77	356	3.64
5	345	4.77	355	4.62	355	4.29
6	349	5.65	360	5.45	352	4.92
7	350	6.56	356	6.25	358	5.88
8	348	7.37	360	7.00	354	5.94
9	347	8.23	359	7.72	361	6.78
10	350	9,01	359	8.34	355	7.71

Table 6.9: Cost and speedup for Graph S50_5

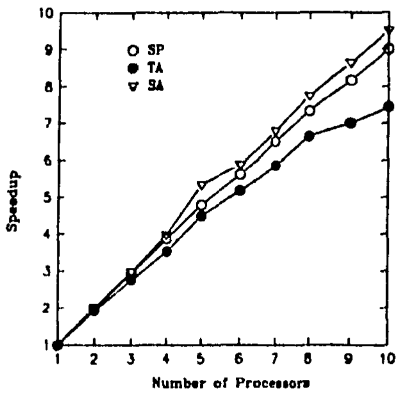

Figure 6.9: Speedup for graph *Line*60_6 with up to 10 processors

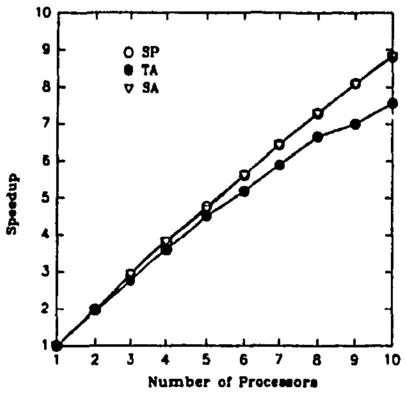

Figure 6.10: Speedup for graph *Ring*60_6 with up to 10 processors

6.3 Observation

From the above experiments, we can make the following observations:

1. Our sequential stochastic probe always yields the best solutions for all the problem instances with less CPU time for our task assignment on this *nnuniform single-bus completion cost model.*

2. Both the parallel stochastic probe and simulated annealing achieve almost linear speedup for all the problem instances with solution quality comparable to those

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	2596	1.00	2613	1.00	2617	1.00
2	2576	1.97	2595	1.98	2581	1.95
3	2572	2.89	2601	2.94	2640	2.78
4	2583	3.83	2592	3.88	2716	3.52
5	2577	4.73	2587	4.81	2687	4.43
6	2579	5.60	2609	5.73	2800	5.02
7	2575	6.41	2603	6.66	2624	5.82
8	2590	7.18	2622	7.52	2681	6.32
9	2589	7.99	2603	8.40	2599	6.71
10	2597	8.84	2619	9.27	2617	7.18

Table 6.10: Cost and speedup for Graph C60_6

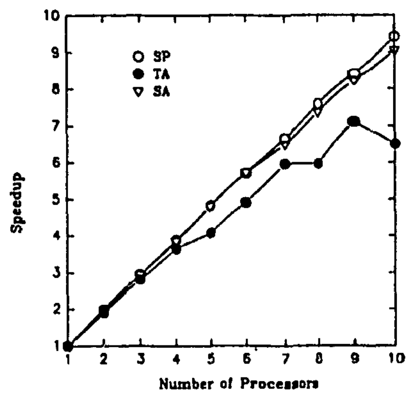

Figure 6.11: Speedup for graph *Mesh*49_6 with up to 10 processors

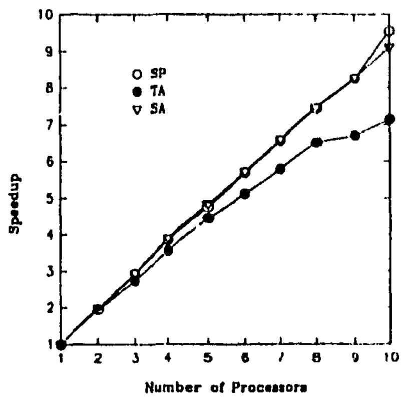

Figure 6.12: Speedup for graph *Tree*60_6 with up to 10 processors

of their sequential versions.

3. Parallel tabu search can also get almost linear speedup with solution quality comparable to those of its sequential version, but the speedup is a little bit worse than that of the parallel SP and SA. This is due to the way we parallelize it (parallelizing the aggressive search at each iteration).

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	465	1.00	468	1.00	463	1.00
2	459	1.98	470	1.97	466	1.95
3	458	2.93	488	2.90	449	2.77
4	457	3.89	488	3.82	458	3.57
5	456	4.80	477	4.68	489	4.48
6	457	5.71	486	5.55	460	5.16
7	459	6.62	489	6.39	464	5.85
8	459	7.51	487	7.17	472	6.33
9	461	8.33	490	7.93	461	6.87
10	460	9.16	494	8.67	472	7.44

Table 6.11: Cost and speedup for Graph S60_6

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	206	1.00	202	1.00	203	1.00
2	199	1.98	204	2.00	201	1.92
3	199	2.94	207	2.98	202	2.76
4	199	3.87	204	3.97	196	3.52
5	199	4.79	202	5.33	208	4.49
6	200	5.61	202	5.88	204	5.17
7	200	6.51	201	6.79	198	5.85
8	201	7.32	203	7.74	196	6.64
9	200	8.17	207	8.64	204	7.00
10	200	9.00	206	9.51	205	7.43

Table 6.12: Cost and speedup for Graph Line60_6

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	447	1.00	435	1.00	443	1.00
2	434	1.99	450	1.97	433	1.95
3	432	2.92	449	2.94	429	2.76
4	431	3.80	447	3.83	435	3.59
5	432	4.77	453	4.71	429	4.51
6	436	5.61	463	5.62	428	5.18
7	436	6.44	453	6.44	456	5.89
8	438	7.28	458	7.30	434	6.66
9	437	8.11	458	8.10	449	7.00
10	439	8.82	455	8.85	432	7.56

Table 6.13: Cost and speedup for Graph Ring60_6

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	183	1.00	186	1.00	187	1.00
2	177	1.97	181	1.98	190	1.89
3	178	2.94	182	2.94	183	2.81
4	176	3.88	186	3.85	190	3.66
5	175	4.83	186	4.81	178	4.08
6	177	5.71	179	5.72	177	4.91
7	175	6.66	189	6.50	187	5.96
8	176	7.59	182	7.37	191	5.96
9	178	8.41	189	8.23	196	7.10
10	176	9.43	185	9.06	207	6.50

Table 6.14: Cost and speedup for Graph Mesh49_6

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	202	1.00	202	1.00	196	1.00
2	196	1.96	198	1.99	207	1.95
3	197	2.93	207	2.94	199	2.75
4	194	3.84	199	3.89	211	3.55
5	195	4.77	200	4.84	210	4.45
6	197	5.70	201	5.72	211	5.12
7	199	6.57	206	6.60	207	5.79
8	200	7.45	204	7.47	202	6.52
9	195	8.26	201	8.28	213	6.72
10	198	9.54	204	9.11	215	7.15

Table 6.15: Cost and speedup for Graph Tree60_6

Graph	Average Cost		
	SP	SA	TA
C30-3	984.3	982.1	990.4
S30-3	253.1	259.3	256.8
C40-4	1478.1	1475.3	1485.9
S40-4	282.5	287.3	281.4
C50-5	1989.5	1995.2	1998.2
S50-5	348.5	355.2	358.5
C60-6	2583.4	2604.4	2656.2
S60-6	459.1	474.9	465.4
Line60-6	200.3	203.9	201.7
Ring60-6	436.2	452.1	436.8
Mesh49-6	177.1	193.1	187.6
Tree60-6	197.3	202.2	207.1

Table 6.16: Average solution cost over all experiments for each benchmark graph with the three parallel algorithms

Chapter 7

General Nonuniform Total Cost Model

In this chapter, we study the General Nonuniform Total Cost Model. The target system for this model is a general heterogeneous workstation farm connected by a interconnection network, in which

- All processors are heterogeneous;
- All processors are connected by an interconnection network;
- Communication between any pair of processes within a processor is negligible;

As a general heterogeneous system, the *execution cost*, *communication cost*, and *interference cost* are three key factors for the performance of the system. Since the interconnection network is used to connect the system instead of a single-bus, all the three costs in the system are processor dependent.

Based on this model, our task assignment problem is to find a mapping $\pi : V \rightarrow [m]$ to minimize the *total cost*

$$cost(\pi) = \sum_{u \in V} X(u, \pi(u)) + \sum_{u \neq v} C(\pi(u), \pi(v), u, v)$$

Our experimental studies on the task assignment for this model is presented as follows. All of the three solution techniques for the problem studied in Chapter 3 are applied. We study both the sequential performance and parallel performance for each of the algorithms.

Details are presented in the Section 7.1 and Section 7.2 respectively. We present our observation for the experiments on this model in the last section.

7.1 Experimental studies with sequential algorithms

We conduct sequential experiments to compare the performances of the three algorithms in sequential version (simulated annealing, tabu search, and our stochastic probe) for the task assignment on this *general nonuniform total cost model.* We run each algorithm 10 times for each benchmark graph (Section 3.5.3).

	C60-6			C80-8			C100-10			C120-12		
	SP	SA	TA	SP	SA	TA	SP	SA	TA	SP	SA	TA
cost	5104	5104	5104	8864	8864	8864	15511	15511	15511	21775	21775	21775
CPU (sec.)	3.3	6.1	1.6	10.8	29.2	5.8	36.7	57.7	13.6	82.8	156.2	29.8

Table 7.1: Performance comparison of sequential algorithms for structured data sets

	S60-6			S80-8			S100-10			S120-12		
	SP	SA	TA	SP	SA	TA	SP	SA	TA	SP	SA	TA
cost	1659	1659	1659	2702	2718	2743	4087	4088	4140	5791	5842	5817
CPU (sec.)	35.2	59.1	41.7	29.0	110.1	54.7	81.0	138.3	113.9	118.3	246.5	123.1

Table 7.2: Performance comparison of sequential algorithms for structured data sets

The experimental results for clustered, sparse, and structured data sets are reported in Table 7.1 to Table 7.3. From these results, we can draw the following conclusions:

1. Compared with SA and TA, the average improvement of SP for solution quality (*cost*) over the 12 problem instances are 0.21% and 0.42% respectively.

	Line60-6			Ring60-6			Mesh49-6			Tree60-6		
	SP	SA	TA	SP	SA	TA	SP	SA	TA	SP	SA	TA
cost	978	986	981	976	977	989	987	788	788	959	959	959
CPU (sec.)	9.5	80.1	18.5	15.1	46.1	52.4	22.3	26.2	27.6	14.6	35.8	23.3

Table 7.3: Performance comparison of sequential algorithms for structured data sets

2. In terms of computation time, the average CPU times for SA and TA are 2.70 and 1.53 times of those for SP respectively.

Our sequential experiments show that, for the task assignment on this model, SP always outperforms SA and TA both in solution quality and running time.

7.2 Experimental studies with parallel algorithms

To evaluate the performances of parallel simulated annealing, parallel tabu search, and parallel stochastic probe for the task assignment on this model, we run each of these three parallel algorithms on 1 to 10 processors for all our benchmark graphs generated for this model.

The experimental results of both solution quality and speedup are reported in Table 7.4 to Table 7.15.

7.2.1 Speedup evaluation

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	5446	1.00	5562	1.00	5104	1.00
2	5398	2.00	5104	2.42	5104	1.24
3	5398	2.50	5104	3.26	5104	1.17
4	5398	2.87	5104	3.94	5104	1.15
5	5398	3.00	5104	4.50	5104	1.04
6	5398	3.20	5104	4.96	5104	1.00
7	5398	3.20	5104	5.33	5104	0.95
8	5398	3.26	5104	5.63	5104	0.97
9	5398	3.33	5104	5.91	5104	0.92
10	5398	3.45	5104	5.97	5104	0.86

Table 7.4: Cost and speedup for Graph C60_6

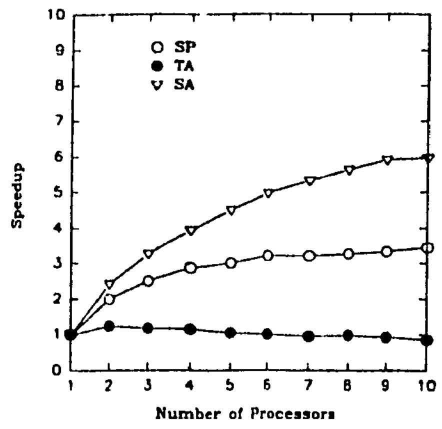

Figure 7.1: Speedup for graph $C60\_6$ with up to 10 processors

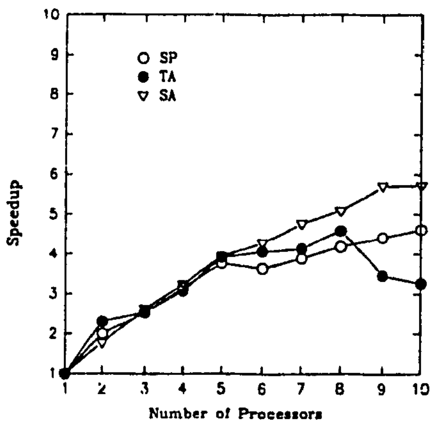

Figure 7.2: Speedup for graph $S60\_6$ with up to 10 processors

Figure 7.1 to Figure 7.12 show the speedup obtained by each of the three parallel algorithms in the 8 problem instances for different types of data sets (clustered, sparse, and structured). From these figures, we can conclude that (with up to 10 processors):

1. Moderate speedups are obtained for these three parallel algorithms. For 10 processors, over 6.63, 5.53, and 3.50 average processor utilization are achieved by parallel SP, SA, and TA respectively.

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	1667	1.00	1659	1.00	1667	1.00
2	1665	2.00	1667	1.77	1664	2.30
3	1666	2.53	1659	2.62	1664	2.54
4	1665	3.14	1660	3.23	1664	3.07
5	1672	3.77	1659	3.94	1659	3.91
6	1677	3.64	1668	4.28	1664	4.07
7	1670	3.90	1689	4.77	1664	4.14
8	1667	4.19	1693	5.09	1673	4.59
9	1666	4.42	1689	5.71	1667	3.47
10	1668	4.62	1686	5.72	1664	3.26

Table 7.5: Cost and speedup for Graph S60_6

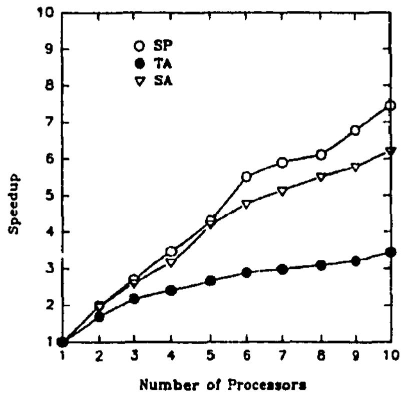

Figure 7.3: Speedup for graph $C80\_8$ with up to 10 processors

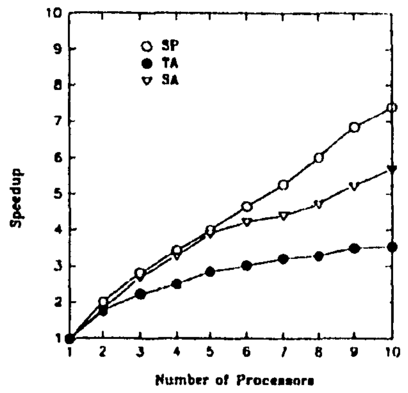

Figure 7.4: Speedup for graph $S80\_8$ with up to 10 processors

2. The speedup in parallel tabu search varies with the problem size. The bigger the problem is, the better speedup the parallel algorithm gets. This is due to the way we parallelize it (parallelizing the aggressive search at each iteration).

3. Parallel simulated annealing achieves better speedup than that of Witte E.E's speculative parallel simulated annealing (refer to Section 3.2.2).

7.2.2 Solution quality comparison

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	8864	1.00	8864	1.00	8864	1.00
2	8864	2.00	8864	1.98	8864	1.70
3	8864	2.70	8864	2.60	8864	2.18
4	8864	3.48	8864	3.18	8864	2.41
5	8864	4.30	8864	4.21	8864	2.67
6	8864	5.50	8864	4.77	8864	2.89
7	8864	5.90	8864	5.13	8864	2.98
8	8864	6.11	8864	5.51	8864	3.09
9	8864	6.78	8864	5.79	8864	3.21
10	8864	7.45	8864	6.21	8864	3.45

Table 7.6: Cost and speedup for Graph C80_8

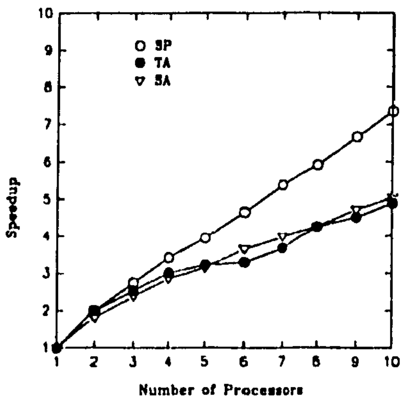

Figure 7.5: Speedup for graph C100_10 with up to 10 processors

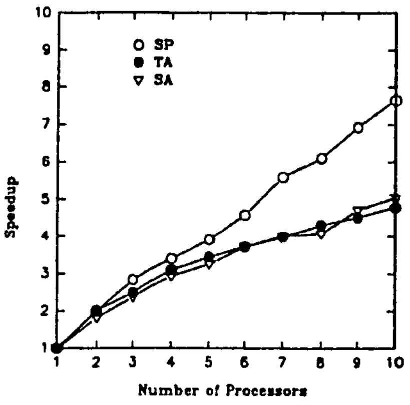

Figure 7.6: Speedup for graph S100_10 with up to 10 processors

Table 7.16 summaries Table 7.4 to Table 7.15 on solution quality by showing the average cost of each algorithm for each problem instance over different number of processors. From this table, we can draw the following conclusions in terms of solution quality:

1. For the 4 cluster data sets, parallel SP improves the solution quality of parallel SA and TA by 0.43% and 0.92% respectively; for the 4 structured data sets, parallel simulated annealing finds solutions with quality 0.37% and 1.87% better than those of parallel SP and TA; for the 4 sparse data sets, all of them obtain

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	2705	1.00	2718	1.00	2743	1.00
2	2704	2.00	2724	1.79	2742	1.74
3	2705	2.81	2745	2.70	2751	2.23
4	2705	3.43	2716	3.29	2749	2.50
5	2707	3.99	2733	3.91	2763	2.84
6	2704	4.65	2721	4.22	2771	3.01
7	2703	5.26	2715	4.41	2774	3.21
8	2706	6.02	2739	4.73	2771	3.29
9	2706	6.84	2729	5.23	2740	3.50
10	2704	7.41	2744	5.57	2746	3.55

Table 7.7: Cost and speedup for Graph S80_8

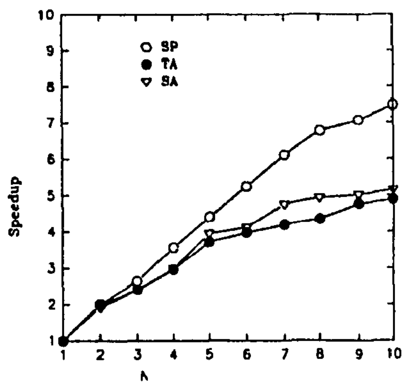

Figure 7.7: Speedup for graph $C120\_12$ with up to 10 processors

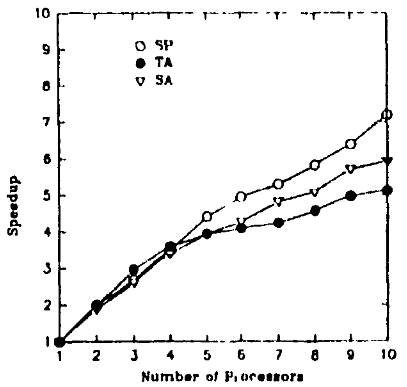

Figure 7.8: Speedup for graph $S120\_12$ with up to 10 processors

the same quality solutions.

2. For all problem instances, the solution obtained from the parallel versions of tabu search, simulated annealing and our stochastic probe are comparable to those of their corresponding sequential versions.

3. For parallel SP, SA, and TA, the average fluctuations for *cost* over different processors for all benchmark graphs are 0.49%, 1.30% and 0.59% respectively.

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	15511	1.00	15511	1.00	15511	1.00
2	15511	2.00	15511	1.82	15511	2.00
3	15511	2.75	15511	2.39	15511	2.54
4	15511	3.42	15511	2.89	15511	3.01
5	15511	3.94	15511	3.14	15511	3.21
6	15511	4.65	15511	3.65	15511	3.29
7	15511	5.38	15511	3.98	15511	3.67
8	15511	5.91	15511	4.25	15511	4.25
9	15511	6.67	15511	4.72	15511	4.51
10	15511	7.35	15511	5.04	15511	4.89

Table 7.8: Cost and speedup for Graph C100_10

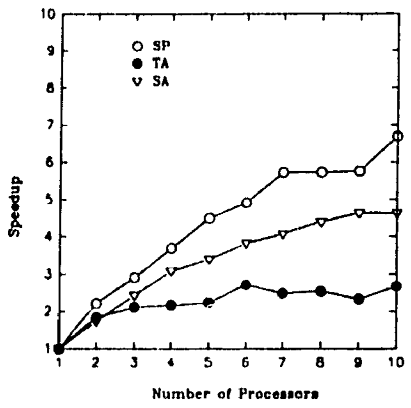

Figure 7.9: Speedup for graph *Line*60_6 with up to 10 processors

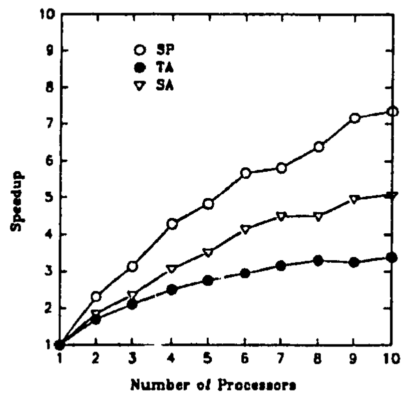

Figure 7.10: Speedup for graph *Ring*60_6 with up to 10 processors

7.3 Observation

From the above experiments, we can make the following observations:

1. Our sequential stochastic probe always yields the best solutions for all the problem instances with less CPU time for our task assignment on this *general nonuniform completion cost model.*

2. Moderate speedup is obtained by the three parallel algorithms with solution quality comparable to those of their sequential versions. Parallel SP obtains

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	4087	1.00	4098	1.00	4125	1.00
2	4095	2.00	4089	1.82	4140	2.01
3	4092	2.83	4088	2.37	4125	2.49
4	4094	3.40	4088	2.94	4125	3.10
5	4095	3.91	4094	3.25	4131	3.45
6	4087	4.58	4089	3.72	4125	3.72
7	4094	5.60	4092	4.01	4137	3.99
8	4089	6.11	4093	4.18	4139	4.32
9	4089	6.93	4095	4.72	4127	4.52
10	4095	7.65	4095	5.04	4127	4.79

Table 7.9: Cost and speedup for Graph S100_10

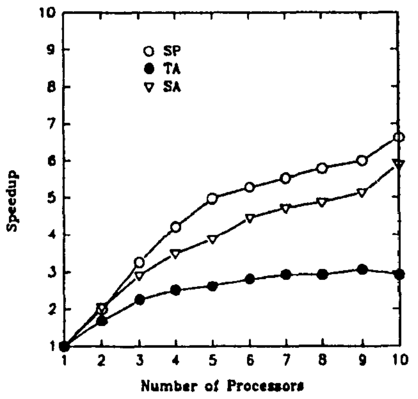

Figure 7.11: Speedup for graph *Mesh*49_6 with up to 10 processors

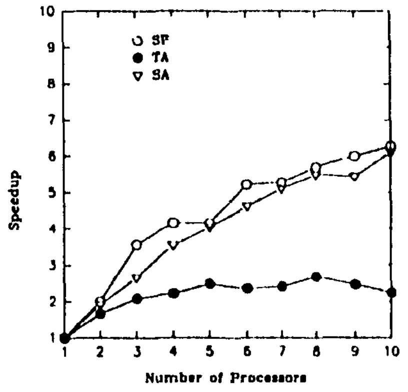

Figure 7.12: Speedup for graph *Tree*60_6 with up to 10 processors

best speedup among the three parallel algorithms.

3. Parallel tabu search can also get almost linear speedup with comparative solution quality to those of its sequential version, but the speedup is a little bit worse than that of the parallel SP, and SA. This is due to the way we parallelize it (parallelizing the aggressive search at each iteration).

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	21775	1.00	21775	1.00	21775	1.00
2	21775	2.00	21775	1.92	21775	2.00
3	21775	2.65	21775	2.41	21775	2.40
4	21775	3.57	21775	2.99	21775	2.97
5	21775	4.40	21775	3.94	21775	3.71
6	21775	5.25	21775	4.12	21775	3.98
7	21775	6.11	21775	4.75	21775	4.20
8	21775	6.78	21775	4.93	21775	4.35
9	21775	7.07	21775	5.01	21775	4.77
10	21775	7.49	21775	5.15	21775	4.89

Table 7.10: Cost and speedup for Graph C120_12

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	5795	1.00	5842	1.00	5850	1.00
2	5795	2.00	5849	1.89	5855	2.00
3	5795	2.70	5849	2.63	5855	2.97
4	5799	3.49	5854	3.42	5855	3.59
5	5798	4.42	5849	3.94	5855	3.94
6	5794	4.94	5847	4.28	5855	4.09
7	5798	5.31	5854	4.83	5863	4.24
8	5798	5.83	5855	5.09	5855	4.59
9	5799	6.40	5857	5.71	5863	4.97
10	5799	7.21	5850	5.93	5863	5.13

Table 7.11: Cost and speedup for Graph S120_12

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	980	1.00	975	1.00	979	1.00
2	982	2.20	981	1.73	979	1.84
3	980	2.91	975	2.42	978	2.11
4	980	3.69	979	3.09	982	2.16
5	981	4.49	977	3.40	979	2.23
6	983	4.91	978	3.82	977	2.71
7	978	5.73	978	4.07	975	2.48
8	985	5.73	978	4.39	977	2.53
9	985	5.77	978	4.63	982	2.33
10	989	6.68	980	4.62	978	2.67

Table 7.12: Cost and speedup for Graph Line60_6

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	978	1.00	976	1.00	987	1.00
2	977	2.30	976	1.84	976	1.69
3	979	3.14	976	2.35	988	2.10
4	977	4.28	976	3.07	976	2.49
5	979	4.83	976	3.52	987	2.74
6	977	5.68	976	4.15	988	2.94
7	977	5.82	976	4.50	987	3.16
8	977	6.40	976	4.51	987	3.32
9	979	7.18	976	4.97	988	3.27
10	983	7.36	976	5.08	987	3.39

Table 7.13: Cost and speedup for Graph Ring60_6e

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	787	1.00	793	1.00	987	1.00
2	789	2.00	787	2.06	987	1.70
3	788	3.26	787	2.91	987	2.25
4	788	4.22	787	3.50	987	2.51
5	791	4.98	787	3.89	987	2.62
6	789	5.28	788	4.46	987	2.80
7	789	5.53	788	4.72	987	2.93
8	792	5.78	798	4.87	987	2.92
9	791	6.00	798	5.13	987	3.06
10	789	6.62	787	5.89	987	2.92

Table 7.14: Cost and speedup for Graph Mesh49_6

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	961	1.00	959	1.00	987	1.00
2	969	2.00	959	1.91	959	1.67
3	968	3.56	959	2.66	964	2.08
4	965	4.16	959	3.55	964	2.22
5	966	4.15	959	4.03	959	2.48
6	963	5.21	959	4.62	964	2.35
7	966	5.27	959	5.10	959	2.40
8	961	5.69	959	5.49	959	2.68
9	966	6.00	964	5.44	982	2.46
10	967	6.28	961	6.11	959	2.24

Table 7.15: Cost and speedup for Graph Tree60_6

Graph	Average Cost		
	SP	SA	TA
C60-6	5402.8	5149.8	5104.0
S60-6	1668.3	1672.9	1665.0
C80-8	8864.0	8864.0	8864.0
S80-8	2704.9	2728.4	2755.0
C100-10	15511.0	15511.0	15511.0
S100-10	4091.7	4092.1	4130.1
C120-12	21775.0	21775.0	21775.0
S120-12	5797.0	5850.6	5856.9
Line60-6	982.3	977.9	978.6
Ring60-6	978.3	976.0	985.1
Mesh49-6	789.3	790.0	987.0
Tree60-6	965.2	959.7	965.6

Table 7.16: Average solution cost over all experiments for each benchmark graph with the three parallel algorithms

Chapter 8

General Nonuniform Completion Cost Model

The workstation farm we study in this chapter is a general heterogeneous system connected by an interconnection network, similar to the one considered in the last chapter. Execution cost, communication cost, and interference cost are all processor-dependent. Our task assignment problem on this kind of workstation farm is to find a mapping π: $[n] \rightarrow [m]$ to minimize the *completion cost*

$$cost(\pi) = \max_{1\leq k\leq m}\{\sum_{\pi(u)=k} X(u,k) + \sum_{\substack{\pi(u)=k \\ \pi(u)\neq\pi(v)}} C(\pi(u),\pi(v),u,v) + \sum_{\substack{\pi(u)=k \\ \pi(v)=k}} C(\pi(u),\pi(v),u,v)\}$$

The following are our experimental studies about task assignment on this model. All the three techniques studied in chapter 3 (both sequential version and parallel version) are studied here.

All the experiments of this model are classified into two parts according to the architecture interconnection network:

1. Experiments on the completely-connected network;

2. Experiments on mesh and hypercube networks.

Our sequential and parallel experimental results are presented in Section 8.1 and 8.2 respectively, followed by Section 8.3, the observation for the experiments.

8.1 Experimental studies with sequential algorithms

Sequential experiments based on the above two categories are conducted to compare the relative performances of these three techniques in sequential versions. We run each algorithm 10 times for each benchmark graph (Section 3.5.3).

8.1.1 Assignments on completely-connected networks

	C30-3			C40-4			C50-5			C60-6		
	SP	SA	TA	SP	SA	TA	SP	SA	TA	SP	SA	TA
cost	1037.7	1051.2	1042.5	1465.7	1492.5	1483.6	1964.2	1968.9	1998.6	2540.0	2579.7	2573.9
CPU (sec.)	0.4	2.1	8.9	2.1	4.6	6.9	2.6	8.6	12.1	17.1	52.4	39.7

Table 8.1: Performance comparisons of sequential algorithms for clustered data sets

	S30-3			S40-4			S50-5			S60-6		
	SP	SA	TA	SP	SA	TA	SP	SA	TA	SP	SA	TA
cost	273.2	293.7	297.5	315.5	319.8	322.3	348.0	358.3	365.7	420.3	432.5	422.8
CPU (sec.)	0.6	2.2	1.8	4.5	7.9	12.0	9.4	25.9	15.5	10.8	58.9	49.9

Table 8.2: Performance comparisons of sequential algorithms for clustered data sets

	S30-3			S40-4			S50-5			S60-6		
	SP	SA	TA	SP	SA	TA	SP	SA	TA	SP	SA	TA
cost	189.3	191.7	213.5	192.0	194.6	195.8	173.1	174.8	176.5	186.2	188.9	190.4
CPU (sec.)	15.3	47.1	47.6	14.4	23.4	58.2	4.0	26.1	34.7	20.7	29.7	42.3

Table 8.3: Performance comparisons of sequential algorithms for clustered data sets

The experimental results for clustered, sparse, and structured data sets are reported in Table 8.1 to Table 8.3. From these data, we can draw the following conclusions:

1. For all the 12 problem instances, SP improves the average solution cost by 1.7% and 3.5% for SA and TA.

2. In terms of computation times, the average CPU times for SA and TA are 3.24 and 6.16 times of those for SP respectively.

In summary, the sequential stochastic probe heuristic provides better solution with less CPU time for the task assignment problem on this model.

8.1.2 Assignments on mesh and hypercube networks

We assume the program has $n = 256$ task models, and the system has 64 processors.

	Cluster			Sparse			Mesh			Tree		
	SA	SP	TS	SA	SP	TS	SA	SP	TS	SA	SP	TS
cost	18624.1	20106.2	26853.2	1984.2	2838.8	4381.1	154.1	204.9	648.9	96.1	124.2	746.8
CPU (sec.)	236.8	220.1	319.9	257.5	206.5	239.8	253.4	258.2	240.2	242.2	246.3	241.2

Table 8.4: Performance comparisons on mesh networks

	Cluster			Sparse			Mesh			Tree		
	SA	SP	TS	SA	SP	TS	SA	SP	TS	SA	SP	TS
cost	9545.8	13130.4	15126.2	1188.3	1248.2	2672.6	122.6	134.4	648.8	79.2	94.1	641.3
CPU (sec.)	167.8	170.3	160.2	296.2	205.5	240.0	273.2	265.3	240.5	350.6	350.5	344.4

Table 8.5: Performance comparisons on hypercube networks

The experimental results for various types of data sets (clustered, sparse, mesh, and tree) are reported in Table 8.4 and Table 8.5.

Among the three heuristics, SA always outperforms SP and TA in solution quality with nearly equal CPU time. Compared with SP and TA, the average improvements of SA over the 8 problem instances are 17.6% and 64.2% respectively.

8.2 Experimental studies with parallel algorithms

Parallel experiments are conducted to evaluate the performance of parallel simulated annealing, parallel tabu search, and parallel stochastic probe for task assignment on this model. They are also divided into two parts according to the interconnection network architecture: the completely-connected networks and mesh and hypercube networks. We run each algorithms on 1 to 10 processors for all our benchmark graphs generated for this model (Section 3.5.3).

8.2.1 Assignments on completely-connected networks

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	1050	1.00	1062	1.00	1076	1.00
2	1051	1.93	1076	1.99	1076	1.94
3	1052	2.86	1062	2.95	1076	2.69
4	1053	3.72	1070	3.92	1076	3.12
5	1050	4.40	1073	4.85	1076	3.87
6	1059	5.52	1077	5.73	1076	4.93
7	1055	6.31	1073	6.59	1076	4.80
8	1058	7.02	1072	7.40	1076	5.24
9	1058	8.07	1065	8.19	1076	6.50
10	1057	8.79	1062	8.88	1076	6.81

Table 8.6: Cost and speedup for Graph $C30\_3$

Table 8.6 to Table 8.17 shows all the experimental results in terms of solution cost and speedup.

Speedup evaluation

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	270	1.00	292	1.00	286	1.00
2	271	1.93	282	2.00	273	1.89
3	272	2.82	273	2.96	287	2.67
4	272	3.71	270	3.92	273	3.06
5	274	4.62	278	4.87	280	3.73
6	273	5.06	278	5.73	273	4.78
7	271	6.06	274	6.58	286	4.58
8	272	6.91	278	7.37	286	5.09
9	273	7.40	282	8.21	287	5.84
10	277	8.62	278	8.90	286	6.31

Table 8.7: Cost and speedup for Graph $S30\_3$

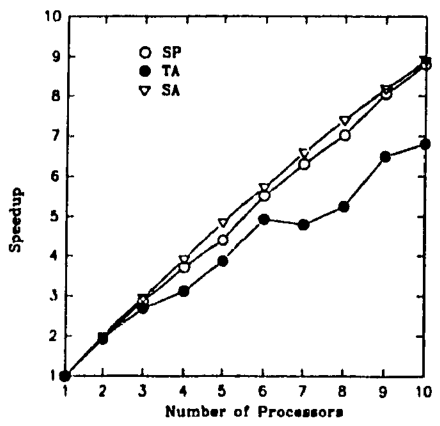

Figure 8.1: Speedup for graph $C30\_3$ with up to 10 processors

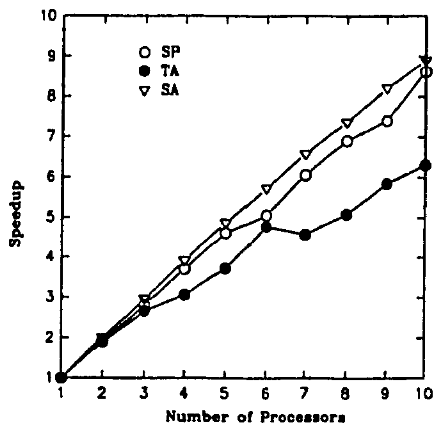

Figure 8.2: Speedup for graph $S30\_3$ with up to 10 processors

To evaluate the speedup from these experiment results, we summarize them in Figure 8.1 to Figure 8.12. For all the problem instances, all of these three parallel algorithms achieve almost linear speedup, with parallel TA's speedup a little bit worse. Processor utilization decreases while the processor number increases. With 10 processors, the average speedup for SP, SA and TA are 8.88, 9.24 and 6.90 respectively.

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	1469	1.00	1472	1.00	1470	1.00
2	1464	1.92	1491	1.98	1470	1.94
3	1471	2.83	1501	2.95	1470	2.71
4	1475	3.68	1487	3.86	1445	3.49
5	1469	4.58	1490	4.77	1445	4.06
6	1471	5.50	1489	5.69	1470	4.69
7	1469	6.24	1539	6.48	1470	5.37
8	1469	7.05	1518	7.28	1445	6.17
9	1471	7.85	1507	8.08	1470	5.76
10	1475	8.61	1506	8.78	1445	6.12

Table 8.8: Cost and speedup for Graph $C40\_4$

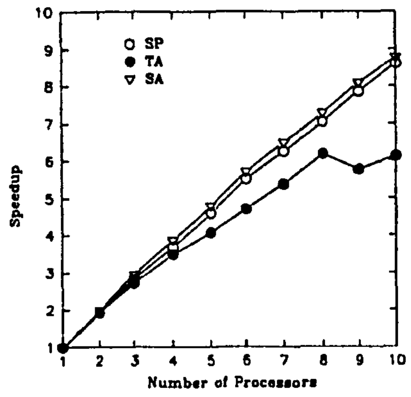

Figure 8.3: Speedup for graph $C40\_4$ with up to 10 processors

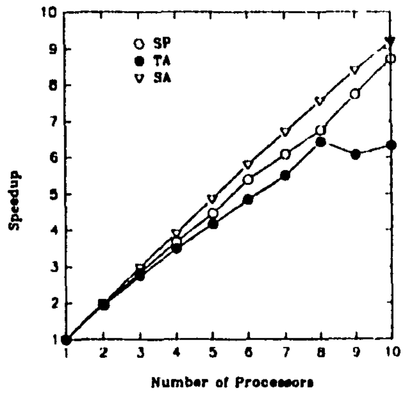

Figure 8.4: Speedup for graph $S40\_4$ with up to 10 processors

Solution quality comparison

From Table 8.6 to Table 8.17, we can draw the following conclusions on the solution quality:

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	321	1.00	318	1.00	310	1.00
2	318	1.93	325	1.99	320	1.95
3	319	2.83	326	2.98	310	2.74
4	316	3.67	317	3.91	317	3.49
5	317	4.45	322	4.87	318	4.16
6	318	5.38	321	5.80	315	4.84
7	318	6.08	325	6.72	318	5.50
8	317	6.76	325	7.56	317	6.43
9	323	7.74	317	8.41	314	6.07
10	324	8.72	320	9.21	314	6.32

Table 8.9: Cost and speedup for Graph $S40\_4$

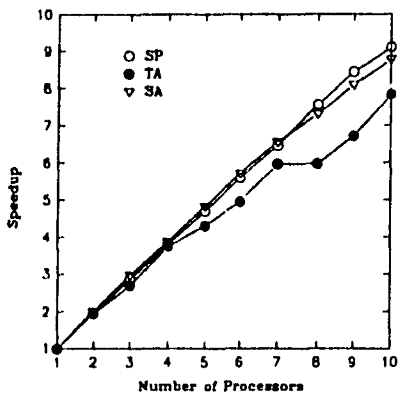

Figure 8.5: Speedup for graph $C50\_5$ with up to 10 processors

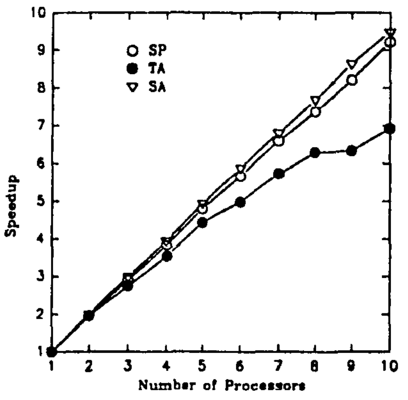

Figure 8.6: Speedup for graph $S50\_5$ with up to 10 processors

1. For all the problem instances, parallel stochastic probe can always outperform the parallel simulated annealing and tabu search in terms of solution quality. Table 8.18 represents the average cost obtained from the three parallel algorithms over the eight benchmark graphs.

2. For all problem instances, the solutions obtained from the parallel versions of tabu search, simulated annealing and our stochastic probe are comparable to those of their corresponding sequential versions.

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	1984	1.00	2006	1.00	2000	1.00
2	1979	1.94	1993	2.00	1963	1.93
3	1983	2.86	2052	2.97	2020	2.70
4	1999	3.79	1980	3.88	1988	3.74
5	1990	4.69	2012	4.82	1983	4.30
6	1986	5.59	2004	5.71	1966	4.95
7	2000	6.46	2000	6.55	1989	5.96
8	2006	7.56	1993	7.31	1964	5.97
9	2001	8.44	2003	8.10	1995	6.71
10	1988	9.09	2021	8.77	1958	7.84

Table 8.10: Cost and speedup for Graph $C50\_5$

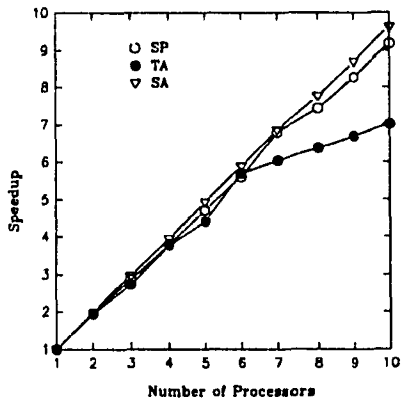

Figure 8.7: Speedup for graph $C60\_6$ with up to 10 processors

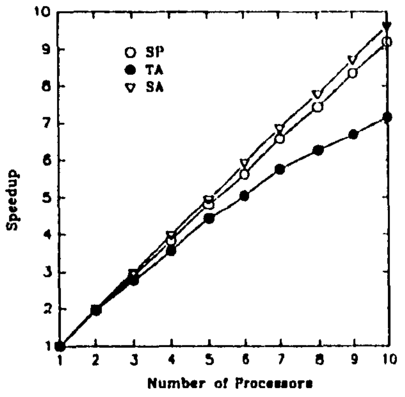

Figure 8.8: Speedup for graph $S60\_6$ with up to 10 processors

3. For parallel SP, SA, and TA, the average fluctuations for cost over different processors for all the benchmark graphs are 2.36%, 4.22% and 4.59% respectively.

8.2.2 Assignments on mesh and hypercube networks

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	360	1.00	369	1.00	459	1.00
2	355	1.97	369	2.00	428	1.95
3	354	2.90	360	2.98	436	2.75
4	352	3.83	365	3.94	440	3.53
5	355	4.79	366	4.92	424	4.43
6	359	5.66	357	5.87	428	4.98
7	359	6.60	354	6.80	429	5.73
8	356	7.37	363	7.69	435	6.30
9	359	8.21	365	8.63	428	6.34
10	358	9.21	364	9.48	421	6.92

Table 8.11: Cost and speedup fro Graph *S*50_5

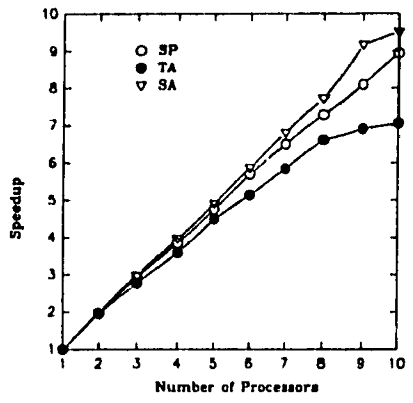

Figure 8.9: Speedup for graph *Li:.*⁻60_6 with up to 10 processors

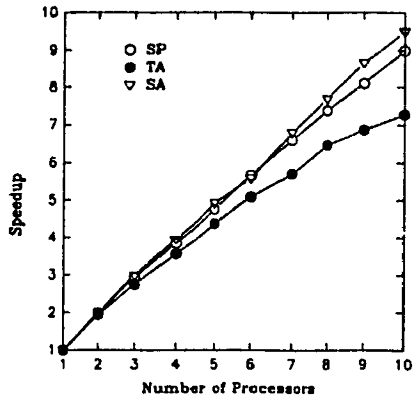

Figure 8.10: Speedup for graph *Ring*60_6 with up to 10 processors

Table 8.19 to Table 8.26 report all experimental results of speedup and cost for all problem instances of the three algorithms solving the task assignment on this model.

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	2557	1.00	2559	1.00	2522	1.00
2	2556	1.95	2552	1.99	2522	1.94
3	2582	2.87	2554	2.98	2522	2.73
4	2569	3.79	2556	3.95	2547	3.78
5	2575	4.72	2574	4.94	2522	4.41
6	2579	5.61	2549	5.89	2536	5.70
7	2571	6.80	2563	6.86	2547	6.04
8	2583	7.45	2566	7.77	2532	6.38
9	2580	8.27	2551	8.70	2534	6.70
10	2576	9.18	1574	9.62	2522	7.02

Table 8.12: Cost ad speedup fro Graph $C60\_6$

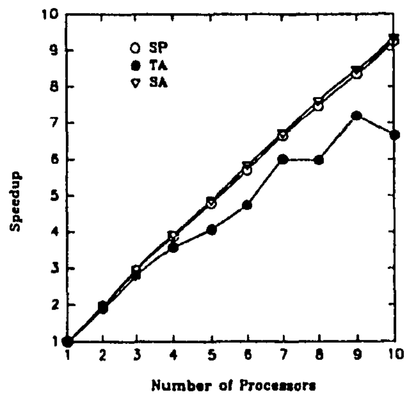

Figure 8.11: Speedup for graph $Mesh49\_6$ with up to 10 processors

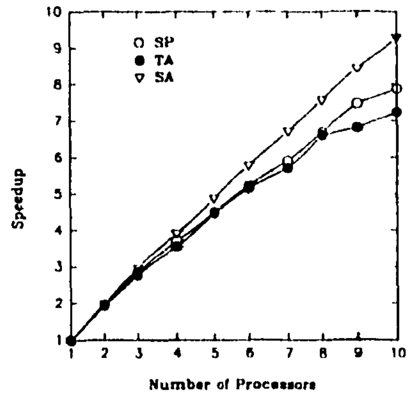

Figure 8.12: Speedup for graph $Tree60\_6$ with up to 10 processors

Speedup evaluation

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	421	1.00	402	1.00	418	1.00
2	423	1.95	418	2.00	425	1.96
3	424	2.89	408	2.99	424	2.76
4	423	3.83	413	3.98	431	3.56
5	425	4.81	407	4.95	417	4.43
6	421	5.62	416	5.92	437	5.04
7	421	6.58	431	6.87	427	5.75
8	424	7.44	420	7.79	417	6.27
9	427	8.34	420	8.72	417	6.69
10	427	9.20	424	9.62	418	7.16

Table 8.13: Cost and speedup for Graph $S60\_6$

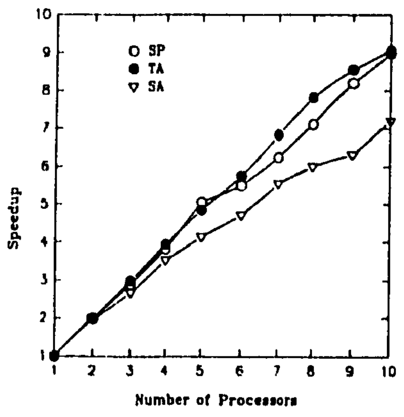

Figure 8.13: Speedup for Graph *cluster* on mesh network with up to 10 processors

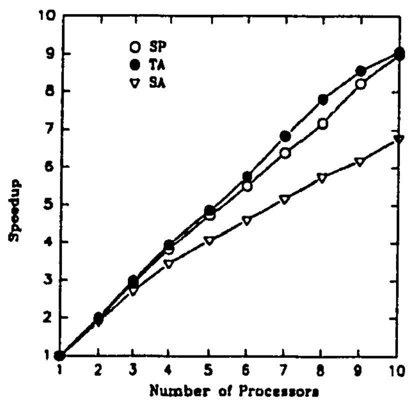

Figure 8.14: Speedup for Graph *sparse* on mesh network with up to 10 processors

Figure 8.13 to Figure 8.20 summaries the performance in terms of speedup from the above Table 8.19 to Table 8.26. From these figures, we can conclude as follows:

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	197	1.00	190	1.00	196	1.00
2	192	1.97	194	1.99	197	1.95
3	190	2.92	193	2.98	194	2.77
4	192	3.84	192	3.96	193	3.58
5	191	4.76	192	4.91	189	4.50
6	193	5.70	192	5.87	198	5.14
7	195	6.50	191	6.81	190	5.84
8	193	7.28	195	7.72	195	6.61
9	197	8.09	194	9.18	197	6.91
10	196	8.95	198	9.52	197	7.06

Table 8.14: Cost and speedup for Graph *Line*60_6

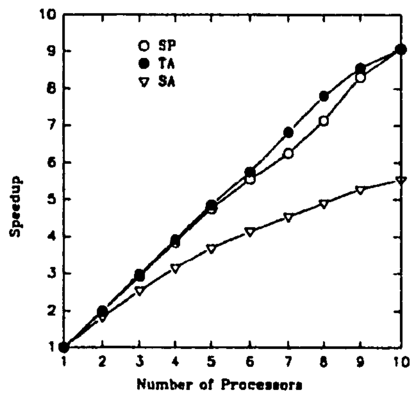

Figure 8.15: Speedup for Graph *mesh* on mesh network with up to 10 processors

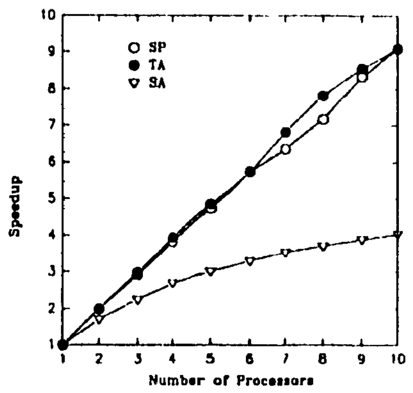

Figure 8.16: Speedup for Graph *tree* on mesh network with up to 10 processors

1. For all the problem instances, all of these three parallel algorithms achieve almost linear speedup, with parallel simulated annealing's speedup a little bit worse, especially for the mesh and hypercube networks. With 10 processors, the average speedup for SP, SA, and TA are 8.99, 5.78 and 9.07.

2. Processor utilization for all the algorithms decreases when more processors are used.

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	195	1.00	193	1.00	192	1.00
2	190	1.98	194	2.00	195	1.94
3	194	2.92	196	2.98	201	2.74
4	191	3.84	193	3.95	185	3.57
5	192	4.76	195	4.92	202	4.38
6	193	5.66	192	5.89	204	5.09
7	190	6.61	196	6.80	193	5.71
8	193	7.39	191	7.70	196	6.47
9	197	8.10	194	8.66	194	6.86
10	195	9.00	201	9.50	195	7.28

Table 8.15: Cost and speedup fro Graph *Ring*60_6

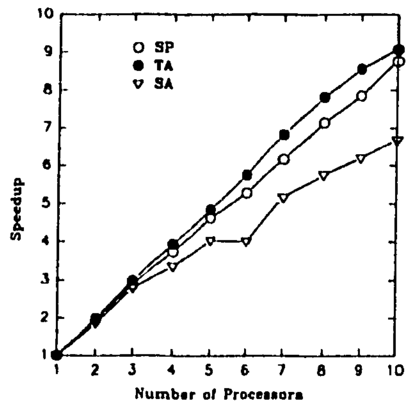

Figure 8.17: Speedup for Graph *cluster* on hypercube network with up to 10 processors

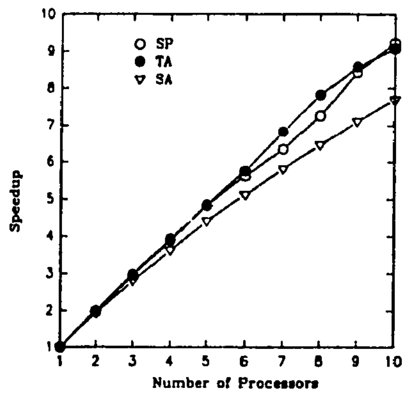

Figure 8.18: Speedup for Graph *sparse* on hypercube network with up to 10 processors

Solution quality

Table 8.27 and Table 8.28 summarize Table 8.19 to Table 8.26 on solution quality by showing the average cost of each algorithm for each problem instance over different number of processors. From these two tables, we can conclude that SA always outperforms SP and TA in solution quality; and compared with SA and TA, the average improvements of SP over the 8 problem instances are 19.53% and 269.29% respectively, and all the solution quality are comparable to those of their sequential

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	176	1.00	172	1.00	172	1.00
2	171	1.98	170	1.99	179	1.90
3	171	2.94	171	2.98	173	2.81
4	174	3.88	173	3.93	183	3.59
5	174	4.80	175	4.88	175	4.08
6	174	5.70	181	5.83	181	4.74
7	169	6.65	174	6.72	166	6.01
8	174	7.47	173	7.61	174	5.98
9	173	8.35	180	8.47	171	7.20
10	174	9.24	173	9.33	169	6.65

Table 8.16: Cost and speedup fro Graph *Mesh*49_6

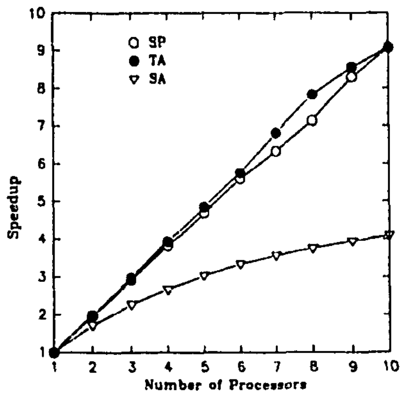

Figure 8.19: Speedup for Graph *mesh* on hypercube network with up to 10 processors

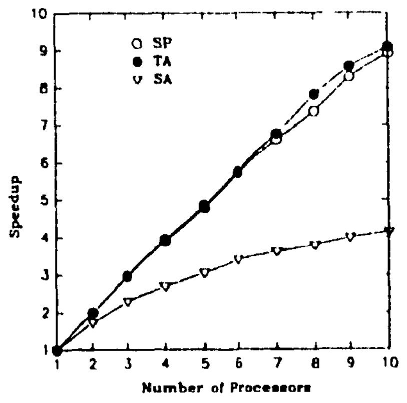

Figure 8.20: Speedup for Graph *tree* on hypercube network with up to 10 processors

versions.

8.3 Observation

From the above experiments, we can make the following observations:

1. Our stochastic probe always yields the best solutions for all the problems on the completely-connected networks with less CPU time, while SA obtains the best solutions for all the problem instances on the mesh and hypercube networks.

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	195	1.00	187	1.00	197	1.00
2	191	1.95	190	1.99	192	1.94
3	194	2.84	185	2.96	186	2.76
4	191	3.71	188	3.92	201	3.56
5	197	4.50	193	4.88	201	4.47
6	190	5.22	193	5.78	194	5.15
7	192	5.91	190	6.72	192	5.71
8	194	6.69	194	7.58	189	6.61
9	201	7.48	195	8.45	195	6.81
10	200	7.89	196	9.28	192	7.25

Table 8.17: Cost and speedup for Graph *Tree60_6*

2. All of parallel SP, SA, and TA obtain almost linear speedup for all problem instances on both the completely-connected networks and mesh and hypercube networks with solution quality comparable to those of their corresponding sequential versions. For parallel SA, a little bit worse speedups are found for mesh and tree graphs on mesh and hypercube networks.

3. Parallel SP obtains the best solutions for all the problem instances on the completely-connected networks, and parallel SA obtains the best solutions for all the problem instances on the mesh and hypercube networks.

In general, regarding the parallel versions, all these three parallel algorithms can achieve almost linear speedup for our task assignment on this *general nonuniform completion cost model.* Parallel SP obtains the best solution quality for problem instances on completely-connected networks while parallel SA provides the best solution quality fro problem instances on the mesh and hypercube networks.

Graph	Average Cost		
	SP	SA	TA
C30-3	1054.3	1069.2	1076.0
S30-3	272.5	278.5	281.7
C40-4	1470.3	1500.0	1463.0
S40-4	319.1	321.6	315.3
C50-5	1991.6	2006.4	1982.6
S50-5	356.7	363.2	432.7
C60-6	2572.8	2559.8	2530.6
S60-6	432.6	415.9	423.1
Line60-6	193.6	193.1	195.6
Ring60-6	193.0	193.5	193.7
Mesh49-6	173.0	172.2	174.2
Tree60-6	194.5	191.1	193.7

Table 8.18: Average solution cost over all experiments for each benchmark graph with the three parallel algorithms

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	21535	1.00	17176	1.00	26789	1.00
2	21597	1.98	17379	1.96	26712	2.00
3	21564	2.90	16489	2.67	26765	2.98
4	21664	3.83	17281	3.53	25517	3.94
5	21732	5.05	16744	4.14	26426	4.85
6	21813	5.50	16492	4.72	25517	5.75
7	21668	6.25	17305	5.55	25517	6.83
8	21902	7.11	17168	6.01	25517	7.82
9	21940	8.21	16207	6.32	25595	8.56
10	21973	8.99	17639	7.20	25517	9.07

Table 8.19: Cost and speedup for cluster graph on mesh networks

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	2809	1.00	2129	1.00	4360	1.00
2	2969	1.96	2163	1.89	4454	2.00
3	2987	2.92	2166	2.71	4747	2.98
4	2998	3.83	2189	3.44	4394	3.94
5	3006	4.73	2154	4.07	4706	4.85
6	3014	5.50	2089	4.59	4359	5.75
7	3024	6.39	2079	5.16	4695	6.83
8	3036	7.17	2161	5.75	4178	7.81
9	3044	8.22	2114	6.18	4178	8.56
10	3055	9.00	2251	6.78	4178	9.07

Table 8.20: Cost and speedup for sparse graph on mesh networks

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	192	1.00	149	1.00	727	1.00
2	191	1.98	155	1.83	771	2.00
3	194	2.93	165	2.54	764	2.98
4	202	3.85	171	3.15	714	3.93
5	203	4.76	179	3.69	737	4.85
6	215	5.55	173	4.14	761	5.75
7	219	6.25	181	4.54	753	6.82
8	225	7.13	175	4.90	722	7.81
9	235	8.30	179	5.27	751	8.54
10	242	9.07	186	5.54	718	9.06

Table 8.21: Cost and speedup for mesh graph on mesh networks

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	115	1.00	107	1.00	744	1.00
2	115	1.98	105	1.71	744	2.00
3	118	2.92	111	2.25	723	2.98
4	118	3.83	117	2.69	726	3.94
5	122	4.74	116	3.01	705	4.85
6	128	5.75	114	3.31	764	5.74
7	127	6.36	111	3.54	717	6.82
8	132	7.16	122	3.71	711	7.81
9	146	8.33	125	3.90	707	8.56
10	144	9.09	126	4.05	711	9.08

Table 8.22: Cost and speedup for tree graph on mesh networks

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	12383	1.00	9863	1.00	15140	1.00
2	12967	1.97	10005	1.87	15140	2.00
3	13300	2.88	10361	2.78	15140	2.98
4	13348	3.75	9604	3.36	15140	3.94
5	13411	4.63	9849	4.04	15140	4.85
6	13472	5.28	9874	4.01	15140	5.75
7	13546	6.17	9630	5.17	15140	6.83
8	13459	7.15	10207	5.76	15140	7.82
9	13672	7.85	9815	6.20	15140	8.56
10	13419	8.77	10158	6.69	15140	9.07

Table 8.23: Cost and speedup for cluster graph on hypercube networks

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	1226	1.00	1190	1.00	2569	1.00
2	1229	1.97	1194	1.93	2567	2.00
3	1241	2.96	1207	2.81	2513	2.98
4	1261	3.89	1210	3.63	2552	3.94
5	1290	4.83	1217	4.41	2552	4.85
6	1323	5.64	1211	5.12	2586	5.76
7	1374	6.35	1223	5.82	2552	6.83
8	1436	7.26	1222	6.64	2514	7.82
9	1482	8.44	1236	7.11	2514	8.57
10	1532	9.22	1240	7.70	2514	9.08

Table 8.24: Cost and speedup for sparse graph on hypercube networks

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	146	1.00	128	1.00	661	1.00
2	143	1.97	133	1.72	658	2.00
3	144	2.93	135	2.28	661	2.98
4	143	3.83	132	2.68	664	3.94
5	147	4.71	138	3.04	669	4.85
6	154	5.60	138	3.34	580	5.75
7	151	6.34	142	3.57	659	6.82
8	155	7.14	139	3.75	566	7.82
9	163	8.30	144	3.95	573	8.56
10	161	9.07	144	4.10	631	9.07

Table 8.25: Cost and speedup for mesh graph on hypercube networks

Proc. No.	SP		SA		TA	
	cost	speedup	cost	speedup	cost	speedup
1	93	1.00	85	1.00	628	1.00
2	91	1.99	83	1.73	633	2.00
3	92	2.95	89	2.29	631	2.98
4	90	3.90	89	2.70	634	3.94
5	90	4.80	91	3.06	655	4.85
6	94	5.71	92	3.40	639	5.76
7	94	6.61	95	3.62	658	6.75
8	95	7.36	95	3.79	647	7.82
9	96	8.28	95	3.99	656	8.56
10	100	8.92	94	4.15	642	9.08

Table 8.26: Cost and speedup for tree graph on hypercube networks

Graph	Average Cost		
	SP	SA	TA
Cluster	21728.8	16988.0	25987.2
Sparse	2994.2	2149.5	4442.7
Mesh	211.8	171.3	741.8
Tree	126.5	115.4	725.2

Table 8.27: Average solution cost over all experiments for each benchmark graph on mesh network with the three parallel algorithms

Graph	Average Cost		
	SP	SA	TA
Cluster	13297.7	9936.6	15140.0
Sparse	1339.4	1215.0	2543.3
Mesh	150.7	137.3	632.2
Tree	93.5	90.8	642.3

Table 8.28: Average solution cost over all experiments for each benchmark graph on hypercube network with the three parallel algorithms

Chapter 9

Conclusion

This thesis studies the task assignment problem for workstation farms.

We first design and propose five task assignment models for workstation farms with different physical configurations: 1) uniform m-way graph partition model, 2) nonuniform single-bus total cost model, 3) nonuniform single-bus completion cost model, 4) general nonuniform total cost model, and 5) general nonuniform completion cost model.

Three effective heuristics based on simulated annealing, tabu search, and stochastic probe are proposed for the task assignment problem. An efficient move set S_3 is designed to combine *vertex move* and *vertex exchange.* To further reduce the computation time, we study the parallelism for these three heuristics. Three effective parallel algorithms are developed based on these three heuristics.

To study the performances of these algorithms (both in sequential versions and parallel versions) on our task assignment problem, we conduct experiments with each algorithm on the five proposed task assignment models. Our experimental results show:

1. Our sequential stochastic probe always yields the best solutions for all the problem instances for the five models with less CPU time.

2. Parallel stochastic probe achieves almost linear speedup for all the problem instances on the five models with solution quality comparable to those of its

sequential version.

3. The speedup of parallel tabu search varies with the problem instances. The bigger the problem instance is, the better speedup the algorithm gets.

4. Parallel simulated annealing obtains speedup better than that of Witte's speculative parallel simulated annealing.

5. For the task assignment on both uniform completion cost model and nonuniform completion cost model, all of these three parallel algorithms achieve almost linear speedup with solution quality comparable to those of their corresponding sequential versions.

6. In general, parallel stochastic probe can get good speedup with high quality solution, and its performance is most stable over different five models.

Bibliography

[1] E. Aarts, F. de Bont, E. Habers, and P. van Laarhoven. Parallel implementations of the statistical cooling algorithm. *Integration, VLSI J.*, pages 209–238, 1986.

[2] E. Aarts and J. Korst. *Simulated Annealing and Boltzmann Machine.* John Wiley and Sons Ltd., 1990.

[3] A. Agarwal. Performance tradeoffs in multithread processors. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):525–539, September 1992.

[4] A. Billionet, M.C. Costa, and A. Sutter. An efficient algorithm for a task allocation problem. *Journal of the Association for Computing Machinery*, 39(3):503–518, July 1992.

[5] Shahid H. Bokhari. Dual processor schedulinmg with dynamic reassignment. *IEEE Transactions on Software Engineering*, SE-5(4):341–589, 1979.

[6] Shahid H. Bokhari. Partitioning problems in parallel, pipelined, and distributed computing. *IEEE Transactions on Computers*, 37(1):48–57, January 1988.

[7] S.Wayne Bollinger and Scott F. Midkiff. Processor and link assignment in multicomputers using simulated annealing. In *Proceedings of International Conference on Parallel Processing*, pages 1–7, 1988.

[8] Andrea Casotto, Fabio Romeo, and Sangiovanni-Vincentelli Alberto. A parallel simulated annealing algorithm for the placement of macro-cells. *IEEE Transactions on Computer-Aided Design*, CAD-6(5):838–847, September 1987.

[9] V. Cerny. Thermodynamical approach to the traveling salesman problem: an efficient simulated annealing algorithm. *Journey of Optimization Theory and Application*, pages 41–51, 1985.

[10] Maw-Sheng Chern, G.H. Chen, and Pangfeng Liu. An lc branch-and-bound algorithm for the module assignment problem. *Information Processing Letters 32*, 32(2):61–71, July 1989.

[11] Yuan-Chien Chow and Walter H. Kohler. Models for dynamic load balancing in a heterogeneous multiple processor system. *IEEE Transactions on Computers*, C-28(5):354–361, May 1979.

[12] Wesley W. Chu. Optimal file allocation in a multiple computer system. *IEEE Transactions on Computers*, C-18(10):885–889, August 1969.

[13] Wesley W. Chu, Leslie J. Holloway, Min-Tsung Lan, and Kemal Efe. Task allocation in distributed data processing. *Computer*, November 1980.

[14] Wesley W. Chu and Kin K. Leung. Module replication and assignment for real-time distributed processing systems. *IEEE*, 75(5):547–562, May 1987.

[15] David T. Connoly. An improved annealing scheme for the qap. *European Journal of Operational Research*, pages 93–100, 1990.

[16] F. Damera, S. Kirkpatrick, and V. A. Norton. Parallel techniques for chip placement by simulated annealing. In *Proceedings of International Conference on Computer Design (ICCD87)*, pages 87–90, October 1987.

[17] David Fernandez-Baca. Allocating modules to processors in a distributed system. *IEEE Transactions on Computers*, 15(11):1427–1436, November 1989.

[18] Armen Gabrielian and Douglas B. Tyler. Optimal object allocation in distributed computer systems. In *Proceedings of 4th IEEE International Conference on Distributed Computing Systems*, pages 88-95, May 1984.

[19] F. Glover. Tabu search - part I. *ORSA Journal on Computing*, 1(3):190-206, 1989.

[20] F. Glover. Tabu search - part II. *ORSA Journal on Computing*, 2(1):4–32, 1990.

[21] V.B. Gylys and J.A. Edwards. Optimal partitioning of workload for distributed systems. *Digest of Papers, CompCon*, pages 353–357, September 1976.

[22] D.S. Johnson, C.R. Aragon, L.A. McGeoch, and C. Schevon. Optimization by simulated annnealing: an exprimental evaluation, part I, graph partitioning. *Operations Reaserach*, 37(6):865-892, Nov.-Dec. 1989.

[23] S. Kirkpatrick, Jr. C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. Research Report RC9355, IBM, 1982.

[24] S. Kirkpatrick, Jr. C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science 220*, pages 671–680, 1983.

[25] S. A. Kravitz and R. Rutenbar. Placement by simulated annealing on a multiprocessor. *IEEE Transactions on Computer-Aided Design*, CAD-6:534–549, July 1987.

[26] C.H. Lee, C.I. Park, and M. Kim. Efficient algorithm for graph-partitioning problem uuing a problem ttansformation method. *Computer-Aided Design*, 21(10):611–618, December 1989.

[27] Cheol-Hoon Lee, Dongmyun Lee, and Myunghwan Kim. Optimal task assignment in linear array networks. *IEEE Transactions on Computers*, 41(7):877–880, July 1992.

[28] Virginia Mary Lo. Algorithm for static assignment and symmetric contraction in distributed computing systems. In *Proceedings of International Conference on Parallel Processing*, pages 239-244, 1988.

[29] Virginia Mary Lo. Heuristic algorithms for task assignment in distributed systems. *IEEE Transactions on Computers*, 37(11):1384 1397, 1988.

[30] Perng-Yi Richard Ma, Edward Y.S. Lee, and Masahiro Tsuchiya. A task allocation model for distributed computing systems. *IEEE Transactions on Computers*, C-31(1):41–47, January 1982.

[31] Camille C. Price. The assignmant of computational tasks among processors in a distributed system. *National Computer Conference*, pages 291–296, 1981.

[32] Camille C. Price and S. Krishnaprasad. Software allocation models for distributed computing systems. In *Proceedings of 4th IEEE International Conference on Distributed Computing Systems*, pages 40–47, May 1984.

[33] Sub Ramakrishnan, Il-Hyung Cho, and Larry A. Dunning. A close look at task assignment in distributed systems. In *Proceedings of 1991 Conference on Computer Communications*, pages 7D.2.1-7D.2.7, 1991.

[34] Sue Ramakrishnan, Larry Dunning, Pratap Thondapu, and Ching Yue. A hybrid task allocation strategy for delay optimization in distributed computer systems. In *Network 92*, 1992.

[35] Gururaj S. Rao, Harold S. Stone, and T.C. Hu. Assignment of tasks in a distributed processor system with limited memory. *IEEE Transactions on Computers*, C-28(4):291–299, April 1979.

[36] A.K. Sarje and G. Sagar. Heuristic model for task allocation in distributed computer systems. *IEE PROCEEDINGS-E*, 138(5), 1991.

[37] Chien-Chung Shen and Wen-Hsiang Tsai. A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion. *IEEE Transactions on Computers*, C-34(3):197–203, 1985.

[38] S. Sofianopoulou. The process allocation problem: a survey of the application of graph-theoretic and integer programming approaches. *Journal of the Operational Research Society*, 43(5):407–413, 1992.

[39] Harold S. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Computers*, SE-3(1):85–93, January 1977.

[40] Harold S. Stone and Shahid H. Bokhari. Control of distributed processes. *Computer*, pages 97–106, July 1978.

[41] L. Tao and Y.C. Zhao. Multi-way graph partition by stochastic probe. *Computers and Operations Reaserach*, 20(3):321–347, 1993.

[42] Ellen E. Witte, Roger D. Chamberlain, and Mark A. Franklin. Parallel simulated annealing using speculative computation. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):483–493, October 1991.

[43] Larray D. Wittie. Computer networks and distributed systems. *IEEE Computer*, pages 67–76, September 1991.