UNIVERSITY AT STONY BROOK

CEAS Technical Report 767

Heuristic Algorithms for Optimizing Computing and Communication Costs for Networked Computer Utilities

Saravut Charcranoon and Thomas G. Robertazzi

Nov. 16, 1998

Heuristic Algorithms for Optimizing Computing and Communication Costs for Networked Computer Utilities

Saravut Charcranoon, Student Member, IEEE

Thomas G. Robertazzi, Senior Member, IEEE

Serge Luryi, Fellow, IEEE

Department of Electrical and Computer Engineering,

University at Stony Brook,

Stony Brook, NY 11794

Phone (516) 632-8400

Fax (516) 632-8494

Email: tom@sbee.sunysb.edu

October 12, 1998

Abstract

A study of using generic heuristic algorithms such as simulated annealing, tabu search as well as problem specific heuristic algorithms to optimize computing and communication costs in a single-level tree (star) network is presented. Two associated problems, i.e., a load distribution sequencing problem and a processor arrangement problem are used to study algorithmic performance. Computational experiments were conducted on randomly generated single-level tree networks. Networks of medium size where optimal solutions have not yet been reported were tested. The results show that simulated annealing and tabu search are competitive approaches to solve these problems. In the load distribution sequencing problem these two algorithms give an impressive best-solution convergence performance although their computational time are not as good as that of heuristic algorithm proposed by the authors. In the processor arrangement problem, simulated annealing and tabu search also exhibit a good performance as far as a quality of final solution is concerned, but do not outperform a heuristic algorithm. This study shows the viability of applying modern heuristic methods to solve problems important for future networked computer utilities and for multiprocessor configuration design.

Keywords Divisible load scheduling, load distribution sequencing, processor arrangement, heuristic algorithms, simulated annealing, tabu search, cost, computer utility, computer configuration design.

1 Introduction

Over the past decades, heuristic algorithms in the area of combinatorial optimization have received a great deal of attention and effort. This is because of their ability to find good or near-optimal solutions within reasonable running times for such problems.

Local search methods, a general class of heuristic algorithms, have also developed into a mature field of combinatorial optimization. The simplest type of local search is a neighborhood search based on a descent method. However a drawback of this kind of algorithm is its propensity to converge to local optima. Over the past 15 years a number of newly developed local search algorithms based on *natural phenomena* such as simulated annealing, tabu search, genetic algorithms, and neural networks have been widely studied and accepted as prospective algorithms to tackle the local optimal problem in combinatorial optimization problems. They provide high quality solutions by being designed to escape from local optimality traps. This is due to some guidance that is incorporated in order to direct a search away from such local optima. Nonetheless, they are not guaranteed to produce optimal solutions. Still they work very well in practice as witnessed by their various applications.

In addition they also possess many advantages including versatility, efficiency and simplicity. A determination from among these modern heuristic methods as to which one is the most appropriate to a particular problem, however, depends on the problem context. That is, there is no consensus on which method is generally the best one in general.

Load scheduling and sharing in parallel and distributed systems in order to optimize the total system resource utilization cost can be formulated as a combinatorial optimization problem. Here a parallelism in a load will be distinguished and exploited to obtained the minimum total cost solutions. One important type of load parallelism is *data parallelism* where data can be arbitrarily divided into smaller parts for processing on different processors. Loads of this type are referred to as *divisible loads.* Example applications include signal processing, image processing, pattern searching, massive computation and simulation programs. We also note that any primitive indivisible load processed frequently enough effectively becomes divisible. To date there has been a good deal of literature on the scheduling of divisible load in parallel and distributed computing systems, for example, as in [3, 4, 5, 7]. The first monograph which treats this topic specifically appears in [6]. However, this prior work does not consider the case when there are *resource utilization* *costs*. That is in fact the main theme of this paper.

In this paper, the problem of optimizing cost incurred from utilizing links and processors to process divisible load in a single-level tree network is considered. Two approaches are proposed in [9, 10] to achieve such a goal: a cost efficient load distribution sequencing approach, and a cost efficient processor arrangement approach. The cost efficient load distribution sequencing problem is to find an order of load distribution from a root node to some or all other nodes in the network such that the total processing cost is as small as possible. The problem is particularly important for future computer utilities. The cost efficient processor arrangement problem is to find a processor arrangement profile by rearranging positions of the processors to receive an assigned fraction of load from the root node so that a minimum total processing cost can be attained. The problem is important in optimizing a multiprocessor system configuration for processing divisible loads. These two methods are different in that the former method involves no physical changes while the latter does.

In [9], the authors proposed a greedy-based load distribution sequencing algorithm which usually provides an optimal solution for small networks. The performance of the algorithm in a medium to large size network, however, was left unexamined due to the unavailability of an efficient optimality verification algorithm. The cost efficient processor arrangement approach was also studied by the same authors in [10]. Therein they proposed the heuristic processor arrangement algorithm which, similarly, almost always attains an optimal solution for small networks. However, they encountered the same limitation in checking the optimality of solutions for medium to large size networks.

This paper focuses on a study of the *comparative performance* of the two main proposed algorithms against simulated annealing and tabu search for the problem of optimizing a computing and communication cost in a single level tree network of medium to large size. Here, simulated annealing and tabu search were selected as modern heuristic algorithms to be studied.

The basic concepts of both simulated annealing and tabu search come from the fields of statistical physics and intelligent problem solving respectively. They were developed to attack complex optimization problems where classical heuristic and optimization methods have failed to be effective and efficient. Simulated annealing is a metaheuristic that guides the local search descent method by emulating a metallurgical cooling process. It is designed to occasionally accept an inferior solution in a controlled maner. Tabu search is also a metaheuristic which shares with the simulated annealing algorithm an ability to direct the local search descent algorithm away from a local optimal trap. This is done by including a short term and/or long term memory into the algorithm. Both of these metaheuristics have proved to yield very satisfactory results in many applications as reported in [1, 18, 21].

This paper is organized as follows. The background of combinatorial optimization and local search are provided in section 2 and 3. An overview of the underlying problems and their formulation are presented in section 4 and 5. In section 6, the basic concepts of simulated annealing and its associated working algorithm are discussed. Similarly, the basic concept and a working algorithm of tabu search is given in section 7. In section 8, performance comparisons drawn from computational experiments are discussed. Finally the conclusion appears in section 9.

2 Combinatorial Optimization

The combinatorial optimization problem is a problem that involves a search for the best solution from a finite or a countably infinite set [20]. Well-known examples are, for example, the traveling salesman problem, the graph coloring problem, the maximum independent set problem, the scheduling problem, and so on. In such problems, a solution could be a graph, an integer number, a set, a permutation, or a sequence. A set of all solutions is typically referred to as a solution space. A cost function value is then defined for every solution. In a combinatorial optimization problem the objective is to choose the minimum or maximum cost solution(s) from the solution space. In terms of mathematical formulation, a combinatorial optimization problem can be summarized as below.

Definition 1 *Given an instance of a combinatorial optimization problem with the solution space $\mathcal{S}$ which is the finite set of all possible solutions and the cost function f which is a mapping defined by $f : \mathcal{S} \rightarrow \Re$*

For minimization, the problem is to find a solution $s^* \in \mathcal{S}$ which satisfies

$$f(s^*) \leq f(s), \quad \text{for all} \quad s \in \mathcal{S}$$

For maximization, the problem is to find a solution $s^* \in \mathcal{S}$ which satisfies

$$f(s^*) \geq f(s), \quad \text{for all} \quad s \in \mathcal{S}$$

Such a solution s^* is called a globally-optimal solution or an optimum, $f^* = f(s^*)$ denotes the optimal cost and $s^* = \{s \in \mathcal{S} | \ f(s) = f^*\}$ is called the set of optimal solutions.

Many combinatorial optimization problems have been shown to be NP-hard problems [12]. That is computational time grows exponentially with problem size. There is no known efficient algorithm to find the optimal solutions for such problems within time that is a polynomial function of the problem size. Nevertheless, these NP-hard problems still need to be "solved". Therefore, to solve such problems in a manageable and acceptable computational time, one may have to be satisfied occasionally with a suboptimal solution.

With this in mind, the task is then reduced to find a method that obtains a good solution that may not be an optimal solution within a reasonable amount of time. Many approaches have been proposed so far. These approaches can generally be classified into two main classes; the enumerative method and the heuristic method. In this paper, the focus is on heuristic methods. A great number of heuristic methods have been developed to find the best possible solution in an efficient way such as the greedy-based methods and the local search-based methods. Local search-based methods include simulated annealing, tabu search, and genetic search.

3 Local Search

Local search algorithms are an important class of heuristic algorithms that relies on an iterative improvement of the cost function by searching over a neighborhood. The use of a local search algorithm requires three main elements: the definition of solutions, a cost function and a neighborhood structure.

Definition 2 *Let $(\mathcal{S}, f)$ be an instance of a combinatorial optimization problem. A neighborhood structure is assumed to be a mapping $\mathcal{N} : \mathcal{S} \rightarrow 2^{\mathcal{S}}$ which defines for each solution $s \in \mathcal{S}$ a set $\mathcal{N}(s) \subseteq \mathcal{S}$ of solutions and each $s' \in \mathcal{N}(s)$ is called a neighbor of s.*

Definition 3 *Let $(\mathcal{S}, f)$ be an instance of a combinatorial optimization problem and $\mathcal{N}$ a neighborhood structure. Each solution $s' \in \mathcal{N}(s)$ can be reached directly from s by an operation called a move, and s is said to move to s' when such an operation is performed.*

A local search starts off with a given initial solution. By applying a move operation, it attempts to find a better solution by searching a neighborhood of a current solution. If such a solution is found, it replaces the current solution. Otherwise, the algorithm stops at the current solution, which is a local optimum.

Definition 4 *Let $(\mathcal{S}, f)$ be an instance of a combinatorial optimization problem and $\mathcal{N}$ a neighborhood structure, then $\hat{s} \in \mathcal{S}$ is called a locally optimal solution or a local optimum with respect to $\mathcal{N}$ if,*

for a minimization problem,

$$f(\hat{s}) \leq f(s), \qquad \forall s \in \mathcal{N}(\hat{s}),$$

for a maximization problem:

$$f(\hat{s}) \geq f(s), \qquad \forall s \in \mathcal{N}(\hat{s})$$

This method is also referred to as a neighborhood search (NS). It should be noted here that the term local search and neighborhood search will be used interchangeably throughout this paper. A description of a basic neighborhood search for a minimization problem can be given below.

<u>A Basic Neighborhood Search Method</u>

```
Step 1.    Start with an initial solution $s_0 \in \mathcal{S}$.
           Set $s = s_0$, where $s$ is a current solution.
           Set $s^* = s$, where $s^*$ is the best solution so far.
```

```
          Set $f^* = f(s^*)$, where $f^*$ is the best cost so far.
Step 2.   Choose $s' \in \mathcal{N}(s)$ by calling a move operation.
Step 3.   If there exists $s'$ which satisfies the choice criteria and
          no other termination criteria are satisfied, then
          Step 3.1   Set $s = s'$.
          Step 3.2   If $f(s') < f^*$, then
                     Set $s^* = s'$ and $f^* = f(s')$.
                     Go to Step 2.
          Step 3.3.  Otherwise stop.
Step 4.   Output $s^*$ as the approximate to the optimal solution.
```

Here the choice criterion and the termination criteria need to be specified. A descent algorithm, for example, can be obtained by defining the choice criteria and termination as follow,

Descent Algorithm

```
Step 3.   If there exists $s'$ such that $f(s') < f(s)$ then,
          Step 3.1   Set $s = s'$.
          Step 3.2   If $f(s') < f^*$, then set $s^* = s'$ and $f^* = f(s')$.
                     Go to Step 2.
```

It can be seen that the descent algorithm continues searching until no further improvment in the cost is obtained then it terminates. That is the termination criterion. This may happen at a local optimum in which case it causes the algorithm to stop and no longer proceed to find a globally optimal solution.

By defining the choice and termination criteria differently, different algorithms will be established as will be seen later in the case of simulated annealing and tabu search.

The move operation may be, for example, a pairwise interchange operation or an adjacent pairwise swapping to a current solution, or a random selection of a new solution from a defined neighborhood of a current solution.

4 Problem Overview

4.1 Model and Concept

A single-level tree network with $(N + 1)$ processors and (N) links is shown in Figure 1. All the processors are connected to the root processor, p_0, via communication links. Associated with the links and processors are the linear cost coefficients, c_1^l, c_2^l,..., c_N^l and c_0^p, c_1^p, c_2^p,..., c_N^p, respectively as depicted in Figure 2. The root processor, assumed to be the only processor at which the divisible load arrives, partitions a total processing load into $(N+1)$ fractions, keeps its own fraction α_0, and distributes the other fractions α_1, α_2,..., α_N to the children processors p_1, p_2,..., p_N respectively and sequentially. Each processor begins computing immediately after receiving its assigned fraction of load and continues without any interruption until all of its assigned load fraction has been processed.

In order to minimize the processing finish time, all of the utilized processors in the network must finish computing at the same time. The process of load distribution can be represented by Gantt-chart-like timing diagrams, as illustrated in Figure 3. From the timing diagram in Figure 3 the fundamental recursive equations can be formulated as follows:

$$\alpha_i w_i T_{cp} = \alpha_{i+1} z_{i+1} T_{cm} + \alpha_{i+1} w_{i+1} T_{cp} \qquad (1)$$

$$\alpha_{i+1} = k_i \alpha_i = (\prod_{j=0}^{i} k_j)\alpha_0 \quad i = 0, ..., N-1 \qquad (2)$$

Here:

$$k_i = \frac{w_i T_{cp}}{(z_{i+1} T_{cm} + w_{i+1} T_{cp})} \qquad i = 0, ..., N-1$$

$$1 = \alpha_0 + \alpha_1 + ... + \alpha_N \qquad (3)$$

α_i:	The load fraction assigned to the i^{th} processor.
w_i:	The inverse of the i^{th} processor's computing speed.
z_i:	The inverse of the i^{th} link's speed.
T_{cp}:	Time to process an entire load by a standard processor, $w_{standard} = 1$.
T_{cm}:	Time to communicate an entire load by a standard link, $z_{standard} = 1$.

From the recursive equations, equation (1), the closed-form expression of α_0, the fraction of load assigned to the root processor and the other processor load fractions can be obtained:

$$\alpha_0 = \frac{1}{D}\prod_{i=1}^{N}(z_iT_{cm} + w_iT_{cp}) \tag{4}$$

$$\alpha_1 = \frac{1}{D}(w_0T_{cp})\prod_{i=2}^{N}(z_iT_{cm} + w_iT_{cp}) \tag{5}$$

$$\vdots$$

$$\alpha_n = \frac{1}{D}\prod_{i=0}^{n-1}(w_iT_{cp})\prod_{i=n+1}^{N}(z_iT_{cm} + w_iT_{cp}) \tag{6}$$

$$\vdots$$

$$\alpha_N = \frac{1}{D}\prod_{i=0}^{N-1}(w_iT_{cp}) \tag{7}$$

where:

$$D = \prod_{i=1}^{N}(z_iT_{cm} + w_iT_{cp})$$

$$+ \sum_{n=1}^{N} \left(\prod_{i=0}^{n-1} (w_i T_{cp}) \prod_{i=n+1}^{N} (z_i T_{cm} + w_i T_{cp}) \right) \quad (8)$$

4.2 Total Processing Cost

The total processing cost is a cost incurred by a network in processing an entire load. It is a linear addition of all individual link-processor costs incurred by utilizing individual link-processor pairs. Define:

$$C_0 = c_0^p w_0 T_{cp} \quad (9)$$

$$C_n = c_n^l z_n T_{cm} + c_n^p w_n T_{cp} \quad , n = 1, ..., N \quad (10)$$

Here c_n^p and c_n^l is the computing and communication cost per second of the nth processor and the nth link, respectively. Also C_n is the cost of processing the entire of load on the nth processor, and $\alpha_n C_n$ is the cost of processing the assigned fraction of load (α_n) on the nth processor. Note that the units of z_n and w_n are second/load while $\alpha_n T_{cp}$ and $\alpha_n T_{cm}$ are the amounts of computation and communication "load" respectively for the nth link-processor pair. The total cost, C_{total}, is then defined as a summation of the individual processing costs incurred at each link-processor pair. That is:

$$C_{total} = \alpha_0 C_0 + \sum_{n=1}^{N} \alpha_n C_n \quad (11)$$

By substituting α_0 and all α_n from the previous section into equation (11), the total cost expression with the j and $j+1$ terms explicitly shown is as follows [9, 10]:

$$\begin{aligned}
C_{total} \;=\; & \frac{1}{D}\left\{ \prod_{i=1}^{N}(z_iT_{cm}+w_iT_{cp})(c_0^p w_0 T_{cp}) \right. \\
& + \sum_{n=1}^{j-1}\left[\prod_{i=0}^{n-1}(w_iT_{cp}) \prod_{i=n+1}^{N}(z_iT_{cm}+w_iT_{cp})(c_n^l z_n T_{cm} + c_n^p w_n T_{cp})\right] \\
& + \prod_{i=0}^{j-1}(w_iT_{cp}) \prod_{i=j+1}^{N}(z_iT_{cm}+w_iT_{cp})(c_j^l z_j T_{cm} + c_j^p w_j T_{cp}) \\
& + \prod_{i=0}^{j}(w_iT_{cp}) \prod_{i=j+2}^{N}(z_iT_{cm}+w_iT_{cp})(c_{j+1}^l z_{j+1} T_{cm} + c_{j+1}^p w_{j+1} T_{cp}) \\
& \left. + \sum_{n=j+2}^{N}\left[\prod_{i=0}^{n-1}(w_iT_{cp}) \prod_{i=n+1}^{N}(z_iT_{cm}+w_iT_{cp})(c_n^l z_n T_{cm} + c_n^p w_n T_{cp})\right]\right\}
\end{aligned} \tag{12}$$

4.3 Cost Efficient Load Sequencing

Consider a single level tree network, as illustrated in Figure 1, which can be represented by the following ordered set:

$$\Theta = \{p_0, (l_1, p_1), ..., (l_j, p_j), (l_{j+1}, p_{j+1}), ..., (l_N, p_N)\} \tag{13}$$

This indicates that processor p_1 is the first processor that is assigned a fraction of load from p_0 followed by processor p_2 and so on. Link-processor pairs (l_j, p_j) and (l_{j+1}, p_{j+1}) in the ordered set Θ above may not necessarily be physically adjacent. Any change in the ordered set above is equivalent to a corresponding change in the sequence of load distribution. Therefore, sequencing is a mechanism that changes one ordered set to another ordered set. As an example, a result of sequencing by swapping an adjacent pair j and $j+1$ of Θ can be given as,

$$\Theta' = \{p_0, (l_1, p_1), ..., (l_{j+1}, p_{j+1}), (l_j, p_j), ..., (l_N, p_N)\} \tag{14}$$

A cost efficient load distribution sequencing is thus defined as a procedure to search for the minimum total processing cost order of the children link-processor pairs to receive a fraction of load from the root processor. This procedure works through a sequencing mechanism as just mentioned.

It should be noted here that any change due to a sequencing mechanism does not disturb the network topology. There is no physical change taking place. It is in fact a logical change that takes place. The detailed description of this method can be found in [9].

4.4 Cost Efficient Processor Arrangement

A second problem also involves an ordered set representation of a single-level tree network:

$$\Pi = \{p_0, (l_1, p_1), ..., (l_j, p_j), (l_{j+1}, p_{j+1}), ..., (l_N, p_N)\} \quad (15)$$

A processor arrangement determines which processor is connected to l_1, l_2, ..., l_N. A processor arrangement does not change the order of dispatching fractions of load from the root processor to links, i.e., an element l_j associated with each ordered pair is fixed during the course of processor arrangement. That is, the sequence of load distribution from the root processor does not change from the link point of view. The ordered set in this context will be referred to as a processor arrangement profile. Therefore, a processor arrangement is a mechanism to change from one processor arrangement profile to another processor arrangement profile. As shown in Figure 4, Π' is a result of applying a processor arrangement to a pair of processors j and $j+1$ of Π.

$$\Pi' = \{p_0, (l_1, p_1), ..., (l_j, p_{j+1}), (l_{j+1}, p_j), ..., (l_N, p_N)\} \quad (16)$$

In contrast to the sequencing mechanism, a processor arrangement requires a physical change of a link-processor pairs through processor reordering.

Cost efficient processor arrangement is therefore a procedure to obtain a physical arrangement profile of the children processors to connect to the root processor such that the total cost of processing load in the network is minimized. This procedure works based on the processor arrangement described above. The detailed description can be found in [10].

5 Problem Formulation

5.1 Cost Efficient Load Sequencing Problem

5.1.1 Problem Statement

Cost efficient load sequencing is a problem where the goal is to find a minimum total cost load distribution sequence from a root processor to all other processors in a single-level tree network such that the transmission/processing delay incurred is kept as small as possible.

5.1.2 A Combinatorial Optimization Problem Formulation

A problem instance descriptions

An instance of the cost efficient load sequencing problem is described by a network size, i.e., a number of children link-processor pairs (N), and its parameters which are $z_i, w_i, c_i^l, c_i^p, T_{cp}$, and T_{cm}.

A solution definition and a solution space ($\mathcal{S}$)

A solution is defined by an ordered set θ which represents an order of load distribution from a root processor as given in equation (13). Let $\widehat{\theta}$ be a feasible solution, then $\widehat{\theta}$ is a resulting ordered set from applying a permuting operation to the ordered pairs of children link-processors of θ (the θ that excludes p_0). An example of a feasible solution is θ' given in equation (14). A solution space $\mathcal{S}$ is then defined as a set of all possible $\widehat{\theta}$'s.

It can be seen that the solution is of the discrete type and the solution space is countably finite, the size of which is $N!$.

A cost function (f)

A cost function f is defined as the total cost incurred to process load in the network. It is as given in equation (12).

Problem Formulation

Given a network size and its associated parameters, a solution space $\mathcal{S}$, and the cost function f, the problem is to find Θ^*, a minimum total cost load sequence, such that

$$\min_{\forall \theta \in \mathcal{S}} \quad f(\theta | \vec{\alpha}_\theta^*) \tag{17}$$

and

$$\vec{\alpha}_\theta^* \quad \text{satisfies:} \qquad delay(\vec{\alpha}_\theta^* | \theta) \leq delay(\vec{\alpha} | \theta), \qquad \forall \vec{\alpha} \in \Lambda \tag{18}$$

where

$\theta = \{p_0, (l_1, p_1), ..., (l_j, p_j), ..., (l_N, p_N)\}$

$\vec{\alpha} = (\alpha_0, \alpha_1, ..., \alpha_N)$

$\Lambda = \{\ \vec{\alpha} | \alpha_i \in \mathcal{R}^+ \text{ and } \sum_{i=0}^{N} \alpha_i = 1\}$

$delay(\cdot|\cdot)$ is a delay function, which is given as $\alpha_0 w_0 T_{cp}$.

Thus we seek to find the minimal cost load distribution sequence such that solution time (delay) is minimized for each possible sequence. There are certainly alternate ways of formulating this dual criteria (cost, solution time) problem but we believe that this approach is natural.

Note that given any sequence θ, $\vec{\alpha}_\theta^*$ is obtained from the set of equations given by equation (4)-(7).

5.1.3 The Starting Point

A starting point which consists of a number of children link-processor pairs and all its parameters, i.e., $z_i, w_i, c_i^l, c_i^p, T_{cp}$, and T_{cm}, is randomly generated. The initial θ is determined by the order of children pairs generated.

5.2 Cost Efficient Processor Arrangement Problem

5.2.1 Problem Statement

Cost efficient processor arrangement is a problem where the goal is to find a minimum total cost processor arrangement in a single level tree network for processing an arriving load such that the transmission/processing delay incurred is kept as small as possible.

5.2.2 A Combinatorial Optimization Problem Formulation

A problem instance descriptions

An instance of the cost efficient processor arrangement problem is described by a network size, i.e., a number of children link-processor pairs (N), and its parameters which are $z_i, w_i, c_i^l, c_i^p. T_{cp}$, and T_{cm}.

A solution definition and a solution space ($\mathcal{S}$)

A solution is defined by an ordered set π which represents a physical arrangement profile of the children processors to connect to a root processor as given in equation (15). Let $\widehat{\pi}$ be a feasible solution, then $\widehat{\pi}$ is a resulting processor arrangement profile from permuting the processor components of the order pair in π. An example of a feasible solution is Π' given in equation (16). A solution space $\mathcal{S}$ is then defined as a set of all possible $\widehat{\pi}$'s.

It can be seen that the solution is of the discrete type and the solution space is countably finite, the size of which is $N!$.

<u>A cost function (f)</u>

A cost function f is defined as the total cost incurred to process load in the network. It is given by equation (12).

<u>Problem Formulation</u>

Given a network size and its associated parameters, a solution space $\mathcal{S}$, and the cost function f, the problem is to find π^*, a minimum total cost processor arrangement profile, such that

$$\min_{\forall \pi \in \mathcal{S}} \quad f(\pi | \vec{\alpha}^*_{\pi}) \tag{19}$$

and

$$\vec{\alpha}^*_\pi \quad \text{satisfies:} \qquad delay(\vec{\alpha}^*_\pi|\pi) \leq delay(\vec{\alpha}|\pi), \qquad \forall \vec{\alpha} \in \Lambda \tag{20}$$

where

$\pi = \{p_0, (l_1, p_1), ..., (l_j, p_j), ..., (l_N, p_N)\}$

$\vec{\alpha} = (\alpha_0, \alpha_1, ..., \alpha_N)$

$\Lambda = \{\ \vec{\alpha}|\alpha_i \in \mathcal{R}^+ \text{ and } \sum_{i=0}^N \alpha_i = 1\}$

$delay(\cdot|\cdot)$ is a delay function, which is $\alpha_0 w_0 T_{cp}$.

Again, we seek to find an arrangement which minimizes cost such that the solution time (delay) is minimized for each possible arrangement. Again, alternate ways of formulating this dual criteria optimization problem are possible, but we believe our approach is a natural one.

Note that given any processor arrangement profile π, $\vec{\alpha}^*_\pi$ is obtained from the set of equations given by equation (4)-(7).

5.2.3 The Starting Point

A starting point which consists of a number of children link-processor pairs and all its parameters, i.e., $z_i, w_i, c_i^l, c_i^p, T_{cp}$, and T_{cm}, is randomly generated. The initial π is determined by the order of children pairs generated.

6 Simulated Annealing

6.1 Background and Concepts

The idea of simulated annealing was first studied by Metropolis [19] in 1953 to simulate the cooling of material in a heat bath, a process called an annealing. Metropolis's algorithm simulates the change of state energy of the system when it is subjected to a cooling process. The structure of the resulting solid depends on the rate of the cooling. If it is cooled down slowly, the ground state is achieved and a pure crystal is formed. On the other hand, if fast cooling or quenching is applied then the resulting solid will possess some defects. Later, Kirkpatrick [17] and Černy [8] applied the Metropolis algorithm to optimization problems.

In Metropolis's simulation the system moves to a new state if energy decreases. Otherwise a new state will be adopted according to some controlled probabilistic functions which depends on the current state of the system. In the Metropolis work such a probabilistic function was taken from statistical thermodynamics which can be given by $p(\delta E) = exp(\frac{-\delta}{kt})$ where δ is a change in state energy, t is temperature and k is Boltzmann's constant. In the context of optimization problems, this algorithm is designed to accept uphill moves (in case of minimization) with a probability just given. It can also be viewed as an enhanced version of the descent algorithm of the local search heuristic method. Uphill or non-improvement moves can be accepted as well as downhill or improvement moves. The problem of being trapped at a local optimal solution, which is inherent in descent algorithms, can then potentially be avoided.

Simulated annealing is regarded as a heuristic strategy. Thus it encompasses not just a single algorithm but a family of algorithms. To implement simulated annealing, several decisions need to be made regarding both the generic and the problem specific decisions.

6.2 Basic Method

Here the annealing algorithm which closely follows the original Metropolis simulation is given. It maintains an analogy with the physical cooling process.

In the simulated annealing basic method, which can be described as a random descent algorithm, the next solution is chosen randomly from a defined neighborhood of the current solution. The cost function of the next solution is then evaluated and compared to that of the current solution. The algorithm will make a move to a new solution if a cost improvement is indicated. The simulated annealing algorithm however differs from the random descent algorithm in that moves that do not improve cost could be accepted according to the defined probability function.

The following provides a description of the basic simulated annealing method. Given a minimization problem with solution space S, cost function f and neighborhood structure N, the basic simulated annealing algorithm can be given as:

<u>The Basic Simulated Annealing</u>

```
Step 1.   Start with an initial solution $s_0$.
          Set $s = s_0$ where $s$ is a current solution.
Step 2.   Get an initial temperature $T > 0$.
Step 3.   Get a temperature reduction function $\alpha$.
Step 4.   Repeat
          Step 4.1 Repeat
                   4.1.1   Randomly select $s' \in \mathcal{N}(s)$.
                   4.1.2   $\delta = f(s') - f(s)$.
                   4.1.3   If $\delta < 0$, then $s = s'$.
```

```
                    Else generate random $x$ uniformly in the range $(0,1)$.
                    If $x < exp(\frac{-\delta}{kt})$ then $s = s'$.
                Until the number of replications at this temperature are satisfied.
        Step 4.2 Reduce temperature, set $T = \alpha(T)$.
        Until stopping condition is true.
Step 5. Return $s$ as the approximate to the optimal solution.
```

6.3 Cooling Schedule

The cooling schedule specifies a set of generic decisions or parameters in order to implement the simulated annealing algorithm. It involves the control parameters that determine an initial temperature, a cooling rate, a temperature length, and the stopping criterion. The following are the list of the parameters in the cooling schedule.

Generic Parameters

1) Initemp : used in determining a starting temperature to the algorithm for the current set of run.

2) Tempfactor : represents a temperature reduction coefficient which is constant and less than unity throughout the run. The next temperature

is then determined by $(Tempfactor * (current\_temperature))$.

3) Sizefactor	: determines a number of trials at each temperature. If NB is an expected neighborhood size, the number of trials is then defined to be $(NB * Sizefactor)$.
4) Cutoff	: specifies a threshold of the accepted moves to be passed at a given temperature before the process of annealing can be terminated. It is especially designed to remove unneeded trials from the beginning of the schedule. This parameter, together with the $(Sizefactor)$, control the number of replications at each temperature or the temperature length L.
5) Minpercent	: is used as a threshold to advance a *freezecount* counter. Each time the specified number of replications at each temperature is completed, the ratio of accepted moves to the trials is compared to $(0.5 * Minpercent)$. If it is less than this, then *freezecount* counter will be incremented by 1.
6) Freeze_Lim	: is used together with the *freezecount* counter to determine whether the annealing is frozen so as to terminate the algorithm.

6.4 Acceptance Probability

In the original, perhaps traditional, simulated annealing algorithm an acceptance probability is given by the Boltzmann distribution in the form of $exp(\frac{-\delta}{kt})$, where δ is the difference of the cost of a randomly picked solution and a current solution. At high temperature, almost all the moves are accepted. When the temperature decreases the uphill moves are more likely rejected, only the moves with a small cost increment are admitted. This closely follows the physical annealing process. It works quite successfully. Nonetheless, in some problems [11, 16] the Boltzmann distribution does not outperform some other distributions. It appears that by using different forms of an acceptance probability function one is able to improve the algorithm performance for the problems considered here. In the case of the cost efficient load sequencing problem and the cost efficient processor arrangement problem, it was found that in using the Boltzmann distribution in simulated annealing, the algorithm did not yield satisfactory solutions compared to the final solutions found by the other proposed algorithms, i.e., tabu search, the greedy load sequencing algorithm, and the heuristic processor arrangement algorithm. However it was found that the acceptance probability in the form of $(\frac{\delta}{t})$ exhibits a more satisfactory quality of a solution (as also the case in [2]). Results showed that an algorithm with this latter function provides a better minimum cost solution within the same fixed length of run time.

The acceptance probability function in this paper, $(\frac{\delta}{t})$, works quite differently from the Boltzmann distribution. New moves with a higher cost increment or more steep uphill moves are more likely to be accepted than the less steep uphill moves. At low temperature an uphill move is even more likely to be accepted in order to escape from a local optimum.

6.5 The Working Simulated Annealing Algorithm

As mentioned earlier simulated annealing is regarded as a heuristic strategy or metaheuristic. Some decisions relevant to the real implemetation of the algorithm need to be made for both the generic decisions and the problem specific decisions. The following gives the details of such decisions for the sequencing and the processor arrangement problems.

6.5.1 Generic Decisions

The generic decisions basically involve the cooling schedule that is mentioned in subsection 6.3. Here the value of each parameter will be given

1. Initemp = 0.6
2. Tempfactor = 0.95
3. Sizefactor = 0.95
4. Cutoff = 0.4
5. Minpercent = 0.1
6. Freeze_Lim = 15

6.6 Problem Specific Decisions

The problem specific decisions are concerned with the solution space, the cost function, and the neighborhood structure. The solution space and the cost function are provided in section 5. The neighbor structure for both problems is identical. That is it is uniform and symmetric, i.e., all solutions have the same number of neighbors, and, if solution s_1 is a neighbor of s_2, then s_2 is also a neighbor of s_1. The same mechanism is used to generate a neighbor for both problems. A neighbor of the solution s is obtained by randomly swapping a pair of the elements in the solution s which is either a pair of link-processors for cost efficient load sequencing or a pair of processors for cost efficient processor arrangement.

6.6.1 Working Algorithm

The outline of the working simulated annealing algorithm in this paper follows the one described in [16] with the main difference being the form of accepteance probability function and the inclusion of a reheat process, a procedure to increase a current temperature. The working algorithm is given as below. Here the champion solution is the minimum total cost solution that has been found so far.

The Working Simulated Annealing

Step 0 Generate the initial instance of the problem randomly.
Compute the initial total cost.

Step 1 Set the current solution (s) and the champion solution (s^*) to the initial solution. Set the current total cost $f(s)$ and

```
          the champion total cost $f(s^*)$ to the initial total cost.
          Compute the neighborhood size $NB$.
Step 2.   Compute the initial temperature $(T)$, $(Initemp * f(s))$.
          Set $freezecount$ to 0.
Step 3.   Do
          Step 3.1 Set $changes = trials = champion\_changes = 0$.
              Step 3.1.1 Do
                  Step 3.1.1.1 Set $trials = trials + 1$.
                  Step 3.1.1.2 Generate a random neighbor $s'$ of $s$.
                               Compute the neighbor total cost $f(s')$.
                  Step 3.1.1.3 Calculate $\delta = f(s') - f(s)$.
                  Step 3.1.1.4 If $\delta < 0$,
                               Set $changes = changes + 1$, and $s = s'$.
                               If $f(s') < f(s^*)$, set $s^* = s'$.
                               and $champ\_changes = champ\_changes +$
                  Step 3.1.1.5 If $\delta \geq 0$
                               Generate random number $r$ in [0,1].
                               If $r \leq \frac{\delta}{T}$, set $changes = changes + 1$, $s$
```

```
                While  $(trials < Sizefactor * NB)$ and
                       $(changes < Cutoff * Sizefactor * NB)$
          Step 3.1.2 If $(champ\_changes > 0)$, set $freezecount = 0$.
          Step 3.1.3 If $\frac{changes}{trials} < Minpercent$,
                Step 3.1.3.1 If $\frac{changes}{trials} < (0.5 * Minpercent)$,
                       Increase the temperature $T = 1.5 * T$.
                       Set $freezecount = freezecount + 1$.
                Step 3.1.3.2 Else reduce the temperature using
                       $(Tempfactor)$.
      While  $(freezecount < Freeze\_Lim)$
Step 4.   Output $s^*$ as an approximate to the optimal solution.
```

7 Tabu Search

7.1 Background and Concepts

Tabu search was first introduced by Glover [13] and Hansen [15] in 1986 to solve problems in combinatorial optimization. Later tabu search received much attention from researchers and practitioners in various disciplines.

Many contributions are reported in [14]. The positive results of several computational experiments have made tabu search an established approximation method. It has been successfully applied to obtain the solutions to various kinds of problems such as the travelling salesman problem, the quardratic assignment problem, the graph coloring problem, the vehical routing problem, scheduling, and VLSI design.

The basic idea of tabu search is derived from the principle of intelligent problem solving with the concept of *learn* and *unlearn* using flexible memory systems. Tabu search is designed to resolve the problem of being trapped at a local optimum by imposing and releasing constraints systematically so as to cross valleys or barriers, or to explore otherwise forbidden regions. It, in fact, can be described as a local search method with an inclusion of an adaptive memory structure to guide the search. By exploring the neighborhood of a current solution, the next move has been made to the best solution obtained in such a neighborhood. In contrast to local search, tabu search continues its searching after it reaches the local optimum by occasionally admitting an uphill move as the current solution even though it is an inferior solution. Tabu search can be regarded as a metaheuristic in the sense that to obtain an optimal solution it consists of iteratively applying a heuristic neighborhood search which has to be implemented specifically for each individual problem. Additionally, other generic decisions, such as tabu list length, a tabu-active tenure, an aspiration level, or an intensification and diversification strategy, need to be determined in accordance with the underlying problems. Thus to implement a tabu search, several decisions need to be made for both generic and problem specific decisions.

7.2 The Basic Method

The basic method of tabu search begins with a local search or a neighborhood search algorithm. In tabu search, it is extended to preconditionally accept uphill moves in order to escape from a local optimum. However, this may lead to a situation, called cycling, in which a search falls back to the previous departure point of the search trajectory. To prevent cycling, which may occur in the following steps during the uphill move, a short term memory, in the form of a tabu list, is employed. A tabu list is used to store the solutions or the attributes of the solutions, which are referred to as the moves, that were recently visited. These moves or solutions will be forbidden from being accepted as the next solution for a specified period of time in the future. Here, it can be seen that the structure of neighborhood to be search is restricted and varied from iteration to iteration. The entries of a tabu list can be either an entire solution or some specific attributes of the solution, e.g., a pair of swapping positions in the sequence or an interchanged pair of links in the graph. By recording the attributes of a solution instead of a solution itself, tabu search has blocked out some other potential solutions or the unvisited solutions so far. To solve this problem, an aspiration level which offers the possibility of overriding the tabu status of the move is incorporated. Therefore a move that is tabu but satisfies the aspiration level can be adapted as a new current solution. Tabu search continues searching until the stopping criteria are met, at which time it terminates. A thorough treatment can be found in [14].

The following outlines the basic algorithm of tabu search. Given a minimization problem with solution space $\mathcal{S}$, cost function f and neighborhood structure $\mathcal{N}$, the basic tabu search algorithm can be given as follows:

The Basic Tabu Search

```
Step 1.    Select an initial solution $s_0$.
           Set $s = s_0$.
```

```
Step 2.    Create an empty tabu list $T$ and set an aspiration level $A$.
Step 3.    Repeat
               Step 3.1 Select the best solution $s'$ which is either not tabu or passes
                        an aspiration level from the neighborhood of $s$, $\mathcal{N}(s)$.
                        Set $s = s'$.
               Step 3.2 If $f(s') < f(s^*)$ where $s^*$ denotes the best solution
                        so far, set $s^* = s'$.
               Step 3.4 Update the tabu list $T$ and the aspiration level $A$.
           Until the stopping conditions are true.
Step 4.    Return $s^*$ as the approximate to the optimal solution.
```

7.3 Extensions and Modifications

7.3.1 Memory and Tabu List

Memory and a tabu list are the cornerstones of tabu search in that they provide the opportunity to search beyond local optima. In tabu search the memory structure is catagorized into two classes: a short term memory and a long term memory. The purpose of short term memory is mainly to intensify a search and to move away from a local optimum. The purpose of a long term memory is to diversify a search to a new region. The short term memory affects the modified neighborhood, $\mathcal{N}^*(s)$, in that it identifies the elements in $\mathcal{N}(s)$, the original neighborhood of s, to be excluded from $\mathcal{N}^*(s)$. Meanwhile the long term memory determines the solutions not ordinarily found in $\mathcal{N}(s)$ to be included in $\mathcal{N}^*(s)$. In this paper, a memory of a change of the solution attributes during the recent past, a so called recency-based memory, is used as a short term memory. A tabu list relevant to this short term memory is therefore referred to as a short term tabu list, T_R. A frequency-based memory, which records the number of iterations a certain attribute has changed or has been found in the solutions, is used as a long term memory. Its associated tabu list is called a long term tabu list, T_F. Both the short term and the long term memory work synergetically in broadening the search horizon.

Recency-Based Memory

In this paper, a pair of link-processors or a pair of processors and their positions, which are to be swapped, serve as the attributes to be recorded in a short term tabu list. This link-processor pair or a pair of processors is then prohibited from being swapped again unless its total cost after doing so meets an aspiration threshold. They are thus made to be a tabu-active element. In order to determine how long an element in the tabu list holds a tabu-active status, a tabu tenure is attached to the elements. It indicates a number of iterations to forbid a tabu-active pair at certain positions from being swapped during the next solution. This value is set to the value of a short term tabu list length, $|T_R|$, for every element in the list. The tabu tenure value is decreased by one in every iteration. A tabu-active pair is moved from the short term tabu list after its tabu tenure value is down to zero.

Frequency-Based Memory

Frequency-based memory helps to enhance a chance of finding a better solution. It is in fact capable of providing some solutions or characteristics which are not ordinarily found in the original neighborhood. In other words, it lends itself to diversify the search to other regions. In this paper, a frequency measure defined by the number of times a particular link-processor pair or processor is found occupying a certain place in the solution is adopted. This frequency-based memory is then used to generate a new starting point for the search. (For this new starting point, for a particular position of the solution it is the link-processor pair or processor with the least occupational frequency among the remaining pairs or processors left to be considered that will assume the position of the new starting point.) Ties are broken by preferring the link-processor pair with a smaller index. This newly constructed starting point is then checked against the long term tabu list. If it is not long term tabu, it will be adopted as a new starting point and the long term tabu list is updated by enlisting this new starting point. Otherwise, it will be shifted one position to the right to create a new starting point and it is again checked with the long term tabu list. The process continues until either a successful starting point generation is achieved or fails. If no successful new starting point is obtained, the algorithm is then terminated. Otherwise the long term tabu list will be updated.

7.3.2 Aspiration Criteria

Aspiration criteria are used to determine when a tabu-active status of a move can be overridden. In this paper, an aspiration criterion removes tabu-active status from the move that yields a total cost less than the best total cost found so far. The aspiration level is initially set to the total cost of the best solution obtained so far and is changed whenever a new best solution is found.

7.4 The Working Tabu Search Algorithm

Similar to the simulated annealing algorithm, the working tabu search algorithm needs a specification of the generic decisions and the problem specific decisions.

7.4.1 Generic Decisions

These decisions are parameters that control searching both for intensification and diversification, and that are concerned with the termination criteria. These parameters are provided in the following.

Nbiter	: a current iteration number.
Bestit	: the most recent iteration number that the best solution is updated.
Long_Term	: an iteration number after which a first uphill exploration is conducted. Used along with *nbiter* to determine when to diversify the search.
Nbmax	: a maximum allowance of iterations that no best solution update is found. If this value is reached, the algorithm will terminate. This parameter is set to 6 for the cost efficient load sequencing problem and 8 for the cost efficient processor arrangement problem.
Diversification	: a long term memory related parameter that determines when the

	algorithm moves to search a new region by generating a new starting point. This threshold is set to 8000 for the cost efficient load sequencing problem and 10000 for the cost efficient processor arrangement problem.
Max_Loop	: an independent termination control parameter which is not related to any events that happened during the search. It determines the maximum number of iterations before the algorithm terminates. This parameter is set to 30000 for the cost efficient load sequencing problem and 60000 for the cost efficient processor arrangement problem.
CEM_exhaust	: a termination variable as a result of the generation of a new starting point. It is set to 1 if such a point can no longer be generated.

7.4.2 Problem Specific Decisions

The solution space and the cost function are the ones described in section 5. Both the cost efficient load sequencing and the cost efficient processor arrangement problem use the same neighborhood structure, which is uniform and symmetric as explained in case of the working simulated annealing algorithm. A neighbor is generated by randomly swapping a pair of link-processor or a pair of processors according to the underlying problem.

7.4.3 Working Algorithm

```
Step 1.  Generate a single-level tree network randomly.
         Set the current solution $s$ and the champion solution $s^*$ to the initial solution.
Step 2.  Set $Nbiter = Bestit = 0$. Set $Long\_Term = 0$. Set $CEM\_exhaust = 0$;
         Set $Aspiration$ to $f(s^*)$.
         Get the neighborhood size of the defined neighbor $\mathcal{N}^*(s)$, i.e., $\mathcal{V}^*$.
Step 3.  Do
         Set $Nbiter = Nbiter + 1$.
         Step 3.1 Search the defined neighborhood $\mathcal{N}^*(s)$ for the minimum
                  total cost sequence $s'$ or the first neighbor sequence
                  with $f(s') < Aspiration$. Set $s = s'$.
                  Set $Long\_Term = Nbiter$ when start the first uphill search.
         Step 3.2 Update the short term tabu list (the recency-based memory).
                  Update the frequency statistics of the swapping pairs.
         Step 3.3 If $f(s') < f(s^*)$, then set $s^* = s'$. $Bestit = Nbiter$.
                  Set $Aspiration$ to $f(s^*)$.
         Step 3.4 Set $Loop = Loop + 1$.
         Step 3.5 If $(Nbiter - Long\_Term \geq Diversification)$ and
```

```
            $(Nbiter - Bestit \le Nbmax)$
            Step 3.5.1 Invoke a long term memory for diversifying the search.
                       Generate a new starting sequence that is not long term tabu.
                       Set it to the current solution. Update the long term tabu list.
            Step 3.5.2 Empty the current short term tabu list.
            Step 3.5.3 Set $Nbiter = Bestit = Long\_Term = 0$.
        While($Nbiter - Bestit \le Nbmax$) and ($CEM\_exhaust \neq 1$) and
            ($Loop \le Max\_Loop$)
Step 4. Output $s^*$ as an approximation to an optimal solution.
```

8 Computational Results

8.1 Experimental Procedure

Program implementations for all the algorithms were coded using C++ language on a Sun SPARCstation 20 with a UNIX environment and with the SunOS 5.4 operating system.

In the experiments, effectiveness (the number of best solution convergences), the quality of solution (percentage deviation from the best solution in the case a near-best final solution is obtained), and computational time (time to attain a final solution) are selected as the performance measures.

Initial experiments were conducted to determine the best parameter settings of simulated annealing and tabu search algorithms as discussed in section 6 and 7. In this tuning phase the results from running the proposed algorithms, the greedy sequencing algorithm and the heuristic processor arrangement algorithm which are described in [9] and [10] respectively, were used as a benchmark in each problem. All algorithms were run on networks of size 30, 35, 40 and 50 nodes to order to obtain the best set of parameter values. To determine the parameter settings both quality of a final solution and its corresponding computational time were taken into account. Although these parameter settings may not be the optimal ones, which is in fact impractical to attain since it requires one to test all combinations, they exhibited the best-achieved performance.

In the experimental phase, the network parameters, i.e., $z_i, w_i, c_i^p, c_i^l, T_{cp}$, and T_{cm}, were randomly generated. They were all generated from the uniform distribution [0,3]. For each network size the experiments were performed for three runs, each run with a different set of random number seeds. For each run of both the sequencing and arrangement problems, 2000 network instances were generated. In the case of computational time measurement the number of network instances for each run was 30.

Since the optimal solutions for the test problems were not known, in the result comparison the solution values obtained by each algorithm were compared against the best solutions that had been found among them.

8.2 Experimental Results

8.2.1 The Cost Efficient Load Sequencing Problem

In these experiments three algorithms were compared: the greedy load sequencing algorithm, the simulated annealing sequencing algorithm, and the tabu search sequencing algorithm. The greedy load sequencing algorithm is as described in [9]. It works using the neighborhood search technique with the best improvement criterion. In [9], the greedy load sequencing algorithm was found to obtain the optimal solutions for all the starting points used when network size was not greater than 8. Beyond this network size, solution optimality verification was not possible due to the burden of computation imposed by the exhaustive search which was used as a checking algorithm.

Since an optimal sequence of a network of size greater than 8 is not known, these three algorithms were run and the best result among them was adopted as the best-known solution. The experiment results show that for a network with size smaller than 20 all three algorithms converge to the same final total cost solutions. For a network with size from 20 to 26, both simulated annealing and tabu search always obtain the best solutions, meanwhile the greedy sequencing algorithm does not. The number of near-best solutions as opposed to the best solutions of the greedy sequencing algorithm, however, is very small, ranging from 0.1% for $N = 20$ to 1% for $N = 26$. Morever a percentage of deviation from the best solution is quite low, i.e., less than 2% as shown in table 1. As for a network of size greater than 26, a problem regarding program implementation arose. All three algorithms have a problem with limited accuracy caused by limited arithematic precision of the total cost (we used double precision floating point variable for the total cost). That is, the algorithm must compare two "costs" that are almost identical to a large number of significant figures. This problem hence leads to an uncertainty regarding a validity of the total cost comparison result used to make a decision concerning a move to the next solution. Therefore no networks with size greater than 26 were reported.

It should be noted here that this problem did not prevail for all the tests of networks of size greater than 26. In fact, there were just some network instances that did have this kind of pathological problem.

In terms of a running time, it is evident from figure 5 that the running time of the greedy load sequencing algorithm is far superior to the running time of the other two algorithms, especially when the network is enlarged. In addition, the simulated annealing sequencing algorithm appears to be faster on average than the tabu search sequencing algorithm when network size is increased. They however show no significant difference in solutions produced when the network size is less than 18.

8.2.2 The Cost Efficient Processor Arrangement Problem

Similar to the previous problem, three algorithms, i.e., the heuristic processor arrangement algorithm, the simulated annealing processor arrangement algorithm, and the tabu search processor arrangement algorithm, were studied and their results were compared. The heuristic processor arrangement algorithm is presented in [10]. It is based on a local search with multi-level neighborhood structure and multiple starting points. Its performance through the computational experiments as reported in [10] indicates a small probability of suboptimal solution convergence when a network size is less than 10, and its efficiency can be bounded by a low-order polynomial in number of children processors for a network of size less than 19. However beyond a network size of 10, we were unable to verify the optimality of the final solution obtained from the heuristic algorithm due to the lack of an efficient optimality verification algorithm. Therefore here, the performance of these three algorithms were tested on the single level tree networks when their size is greater than 10.

Since an optimal solution for each network instance was not available, the best solution among these three solutions therefore served as the best-found solution, and was used to measure the number of near-best solution convergences. The experimental results from running all three algorithms on single level tree networks with a number of children processors varied from 10 to 26 are given in table 2 and 3. It can be seen from the table that the heuristic processor arrangement algorithm exhibits the best performance while tabu search shows the least favorable algorithm in terms of both a number of near-best solutions and a quality of the final solution. Nevertheless, the percentage deviation from the best solution is very small even in the case of tabu search, i.e., in this case it comes within 1%. In addition, in all three algorithm results, the number of near-best solutions tends to increase with network size, while the percentage deviation from the best solution appears to be independent of network size. Again tests on networks of size greater than 26 encounter the previously mentioned problem, i.e., the validity of the total processing cost comparison, as found in the load sequencing problem. Therefore no experiments were reported on networks of size greater than 26.

Figure 6 illustrates the computational time per network instance when the number of children processors in the network was varied from 10 to 26. From this figure, the heuristic processor arrangement shows the best running time performance while tabu search seems to be the least attractive in this sense. However, all of the algorithms are approximately the same in running time. Besides, it can be seen from this figure that the running time of the simulated annealing is roughly the same as that of the heuristic processor arrangement algorithm when the number of children processors is increased. Thus from the computational time aspect, it seems that there is no clear-cut computational advantage of using any specific algorithm even though the heuristic processor arrangement may be a bit more favorable as suggested in figure 6.

9 Conclusions

In this paper, the applications of simulated annealing and tabu search techniques, as metaheuristics, to the solution of cost efficient divisible load sequencing and cost efficient divisible load processor arrangement problems are presented. In each problem, simulated annealing and tabu search were implemented according to the problem specification, and their results were then compared to the results of the greedy load sequencing and the heuristic processor arrangement problem. In this computational experiment, networks with a number of children processors greater than 10 which had not been examined before were used to study the effectiveness and efficiency of these underlying algorithms. It turns out that in the load sequencing problem, the simulated annealing and tabu search algorithms always give the best solutions whereas the greedy load sequencing algorithm gives a small number of near-best solutions each of which have a small percentage deviation from the best solution, i.e., less than 2%. On the other hand, the greedy load sequencing algorithm appears to be better than the other two algorithms in terms of the computational time while simulated annealing is better than tabu search in this category. In the processor arrangement problem, the heuristic processor arrangement algorithm gives the most favorable performance while tabu search shows the least favorable performance in all aspects studied, i.e., the effectiveness, the quality of solution, and the computational time. Nontheless, the quality of solution of all three algorithms are fairly close to one another.

In summary, this study shows that simulated annealing and tabu search are quite suitable to solve the load sequencing and the processor arrangement problems in order to obtain the minimum-possible total processing cost. This is evident when the quality of a final solution is concerned even though their efficiency is less than that of the previously proposed problem specific heuristic algorithms. This study also demonstrates the impressive performance of both proposed problem specific algorithms in case of a larger network size. Fruitful future research could be directed toward improvements in the performance of both simulated annealing and tabu search. Another direction could be to investigate other modern heuristic methods such as genetic algorithm or hybrids of these modern heuristic algorithms.

Table 1: Best Solution Convergence of the Load Sequencing Problem

No. of children processors	Seed Number	Greedy	
		No. Near-Best Solutions	Deviation %
20	1	0	N/A
	2	3	0.9053
	3	1	0.0018
22	1	5	0.1483
	2	3	0.1983
	3	5	0.0590
24	1	9	0.9914
	2	10	0.5793
	3	5	0.2974
26	1	23	2.5574
	2	29	1.1126
	3	20	1.1402

Table 2: Best Solution Convergence of the Processor Arrangement Problem

No. of Children Processors	Seed Number	Heuristic		Simulated Annealing		Tabu Search	
		Near-Best Solutions	Deviation %	Near-Best Solutions	Deviation %	Near-Best Solutions	Deviation %
10	1	0	0.0	123	0.366	221	0.689
	2	2	0.0018	116	0.2736	204	0.616
12	1	1	0.0025	145	0.2195	294	0.4682
	2	2	0.0326	137	0.1467	282	0.4749
	3	2	0.091	123	0.2131	279	0.5502
14	1	10	0.157	155	0.1813	303	0.4661
	2	2	0.0518	167	0.1606	314	0.4625
	3	9	0.0432	161	0.1852	331	0.4451
16	1	10	0.0467	171	0.1088	353	0.3763
	2	5	0.0427	173	0.1392	343	0.4012
	3	8	0.0111	174	0.0828	337	0.3485
18	1	7	0.095	179	0.1319	367	0.3173
	2	8	0.112	187	0.1529	387	0.3425
	3	9	0.011	171	0.0869	325	0.2931

Table 3: Best Solution Convergence of the Processor Arrangement Problem

No. of Children Processors	Seed Number	Heuristic		Simulated Annealing		Tabu Search	
		Near-Best Solutions	Deviation %	Near-Best Solutions	Deviation %	Near-Best Solutions	Deviation %
20	1	7	0.0083	214	0.0774	389	0.2771
	2	9	0.0345	169	0.1346	336	0.2745
	3	4	0.023	207	0.0852	373	0.2686
22	1	20	0.0508	207	0.0912	414	0.2589
	2	12	0.0521	200	0.0712	407	0.2392
	3	11	0.0107	205	0.1192	391	0.2534
24	1	11	0.0068	193	0.1074	404	0.2261
	2	16	0.0190	204	0.0659	385	0.2343
	3	14	0.0122	198	0.0540	363	0.2925
26	1	12	0.0085	218	0.0787	412	0.2186
	2	14	0.0386	202	0.0678	379	0.2693
	3	16	0.0492	188	0.0538	407	0.2572

References

[1] E. Aarts and J.K. Lenstra, Ed., *Local Search in Combinatorial Optimization,* Chichester, England : John Wiley & Sons, 1997.

[2] P. Brandimarte, R.Conterno and P. Laface, "FMS production scheduling by simulated annealing," In *Proceedings of the 3rd International Conference on Simulation in Manufacturing*, pp. 235-245, 1987.

[3] S. Bataineh and T.G. Robertazzi, "Distributed computation for a bus network with communication delays," In *Proceedings of the 1991 Conference on Information Sciences and Systems*, The John Hopkins University, Baltimore, MD., pp. 709-714, March, 1991.

[4] S. Bataineh and T.G. Robertazzi, "Bus oriented load sharing for a network of sensor driven processors," *IEEE Transcations on Systems, Man and Cybernetics,* vol. 21, no. 5, pp. 1202-1205, September, 1991.

[5] V. Bharadwaj, D. Ghose and V. Mani, "Optimal sequencing and arrangement in distributed single-level tree networks with communication delays," *IEEE Transactions on Parallel and Distributed Systems,* vol. 5, no. 9, pp. 968-976, September, 1994.

[6] V. Bharadwaj, D. Ghose, V. Mani and T.G. Robertazzi, *Scheduling Divisible Loads in Parallel and Distributed Systems,* Los Alamitos, California: IEEE Computer Society Press, 1996.

[7] J. Blazewicz and M. Drozdowski, "Scheduling divisible jobs on hypercubes," *Parallel Computing,* vol. 21, pp. 1945-1956, 1995.

[8] V. Černy, "A thermodynamical approach to the traveling salseman problem: an efficient simulated algorithm," *Journal of Optimization Theory and Applications,* vol. 45, pp. 41-55, 1985.

[9] S. Charcranoon, T.G. Robertazzi and S. Luyri, "Optimizing Computing and Communication Costs for Networked Computer Utilities," *submitted for publication.*

[10] S. Charcranoon, T.G. Robertazzi and S. Luyri, "Cost Efficient Processor Arrangement in Single Level Tree Networks," *submitted for publication.*

[11] K.A. Dowsland, "Some experiments with simulated annealing techniques for packing problems," *European Journal of Operational Research,* vol 68, pp. 389-399, 1993.

[12] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness,* New York: W.H. Freeman and Company, 1979.

[13] F. Glover, "Future paths for integer programming and links to artificial intelligence," *Computer & Operations Research,* vol. 13, pp. 533-549, 1986.

[14] F. Glover and M. Laguna, *Tabu Search,* Norwell, Massachusetts : Kluwer Academic Publishers, 1997.

[15] P. Hansen, "The steepest ascent mildest descent heuristic for combinatorial programming," Talk presented at the Congress on Numerical Methods in Combinatorial Optimization, Capri, 1986.

[16] D.S. Johnson, C.R. Aragon, L.A. MeGeoch, C. Schevon, "Optimization by simulated annealing: an experimental evaluation; part I, Graph partitioning," *Operations Research,* vol. 37, no. 6, pp. 865-892, 1989.

[17] S. Kirkpatrick, C.D. Gellat and M.P. Vecchi, "Optimization by simulated annealing," *Science,* vol. 220, pp. 671-680, 1983.

[18] G. Laporte, I.H. Osman, Ed., *Annals of Operations Research 63: Metaheuristics in Combinatorial Optimization,* Amsterdam, The Netherlands : Baltzer Science Publisher, 1996.

[19] N.Metropolis, A.W.Rosenbluth, M.N.Rosenbluth, A.H.Teller and E.Teller "Equation of state calculation by fast computing machines," *Journal of Chemical Physics,* vol. 21, pp. 1087-1092, 1953.

[20] C.D. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity,* Englewood Cliffs, New Jersey: Prentice-Hall, 1982.

[21] V.J. Rayward-Smith, I.H. Osman, C.R. Reeves and G.D. Smith, Ed., *Modern Heuristics Search Methods,* Chichester, England : John Wiley & Sons, 1996.

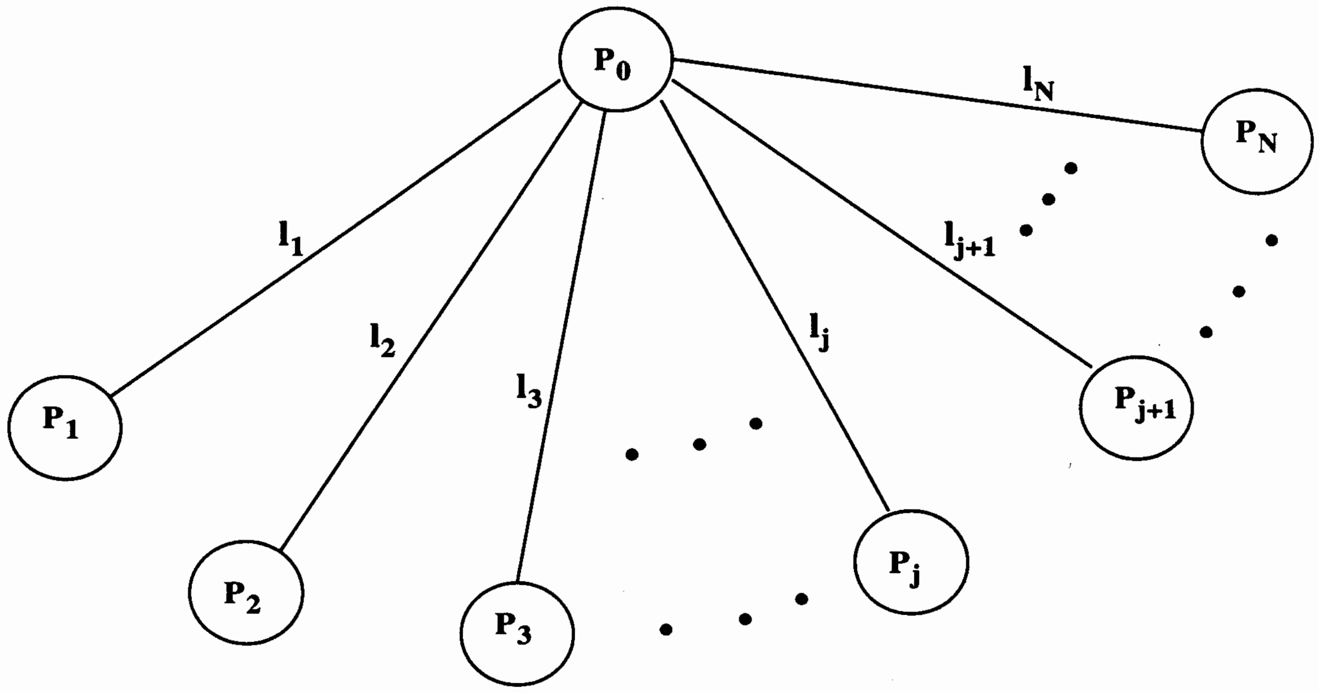

Figure 1: Single level tree network

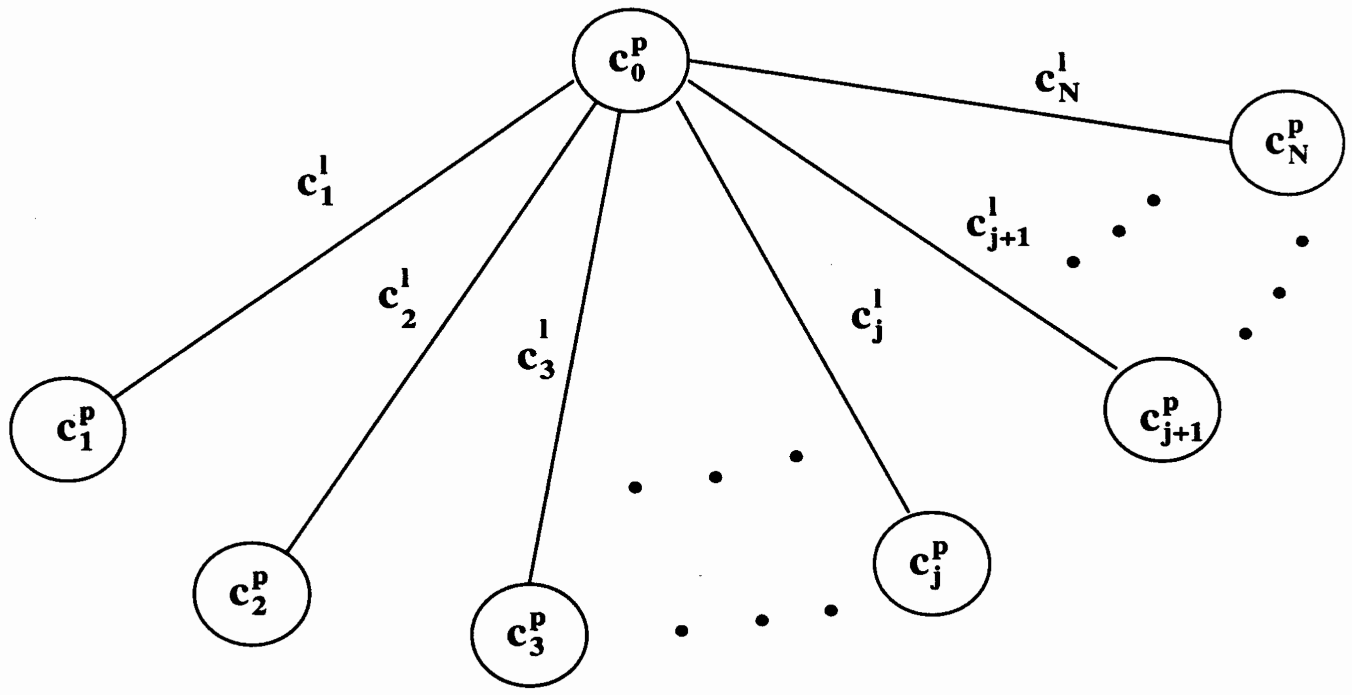

Figure 2: Single level tree network with associated cost coefficients

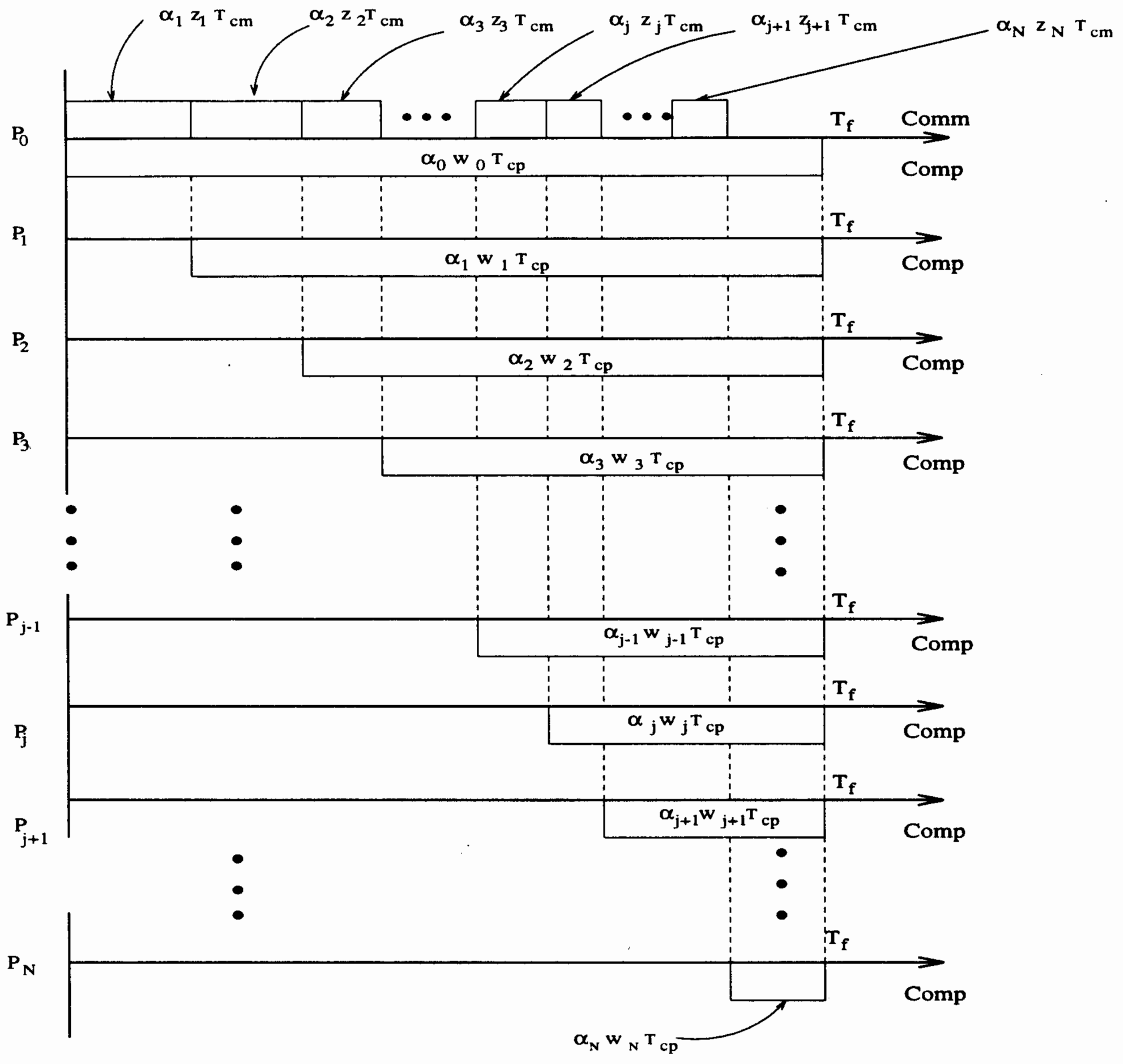

Figure 3: Timing Diagram: Normal Case

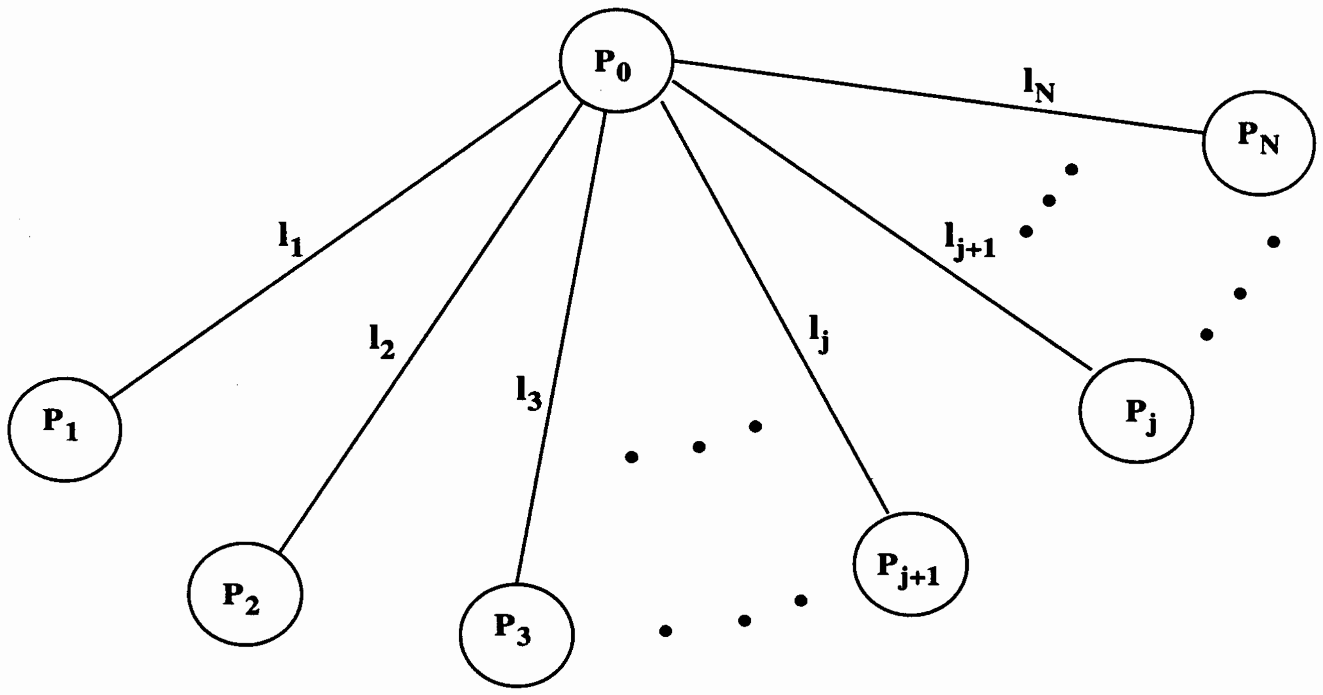

Figure 4: Single level tree network: Adjacent Pairwise Processor Swap

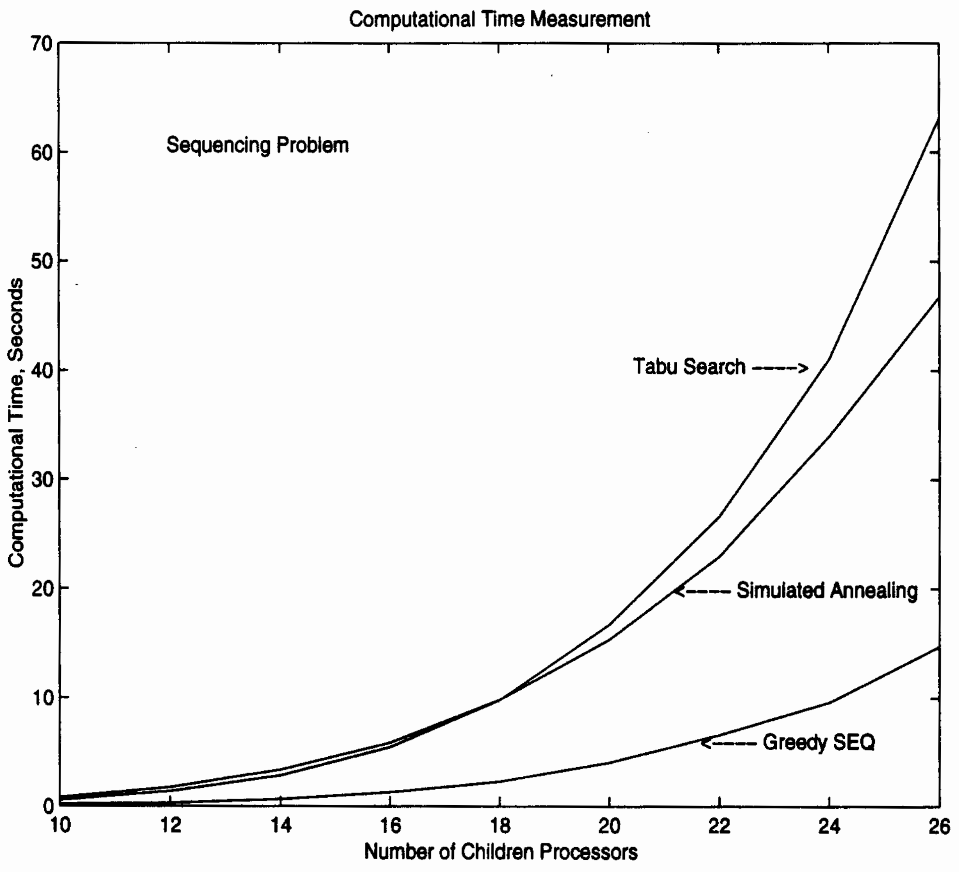

Figure 5: Computational Time of the Load Sequencing Problem

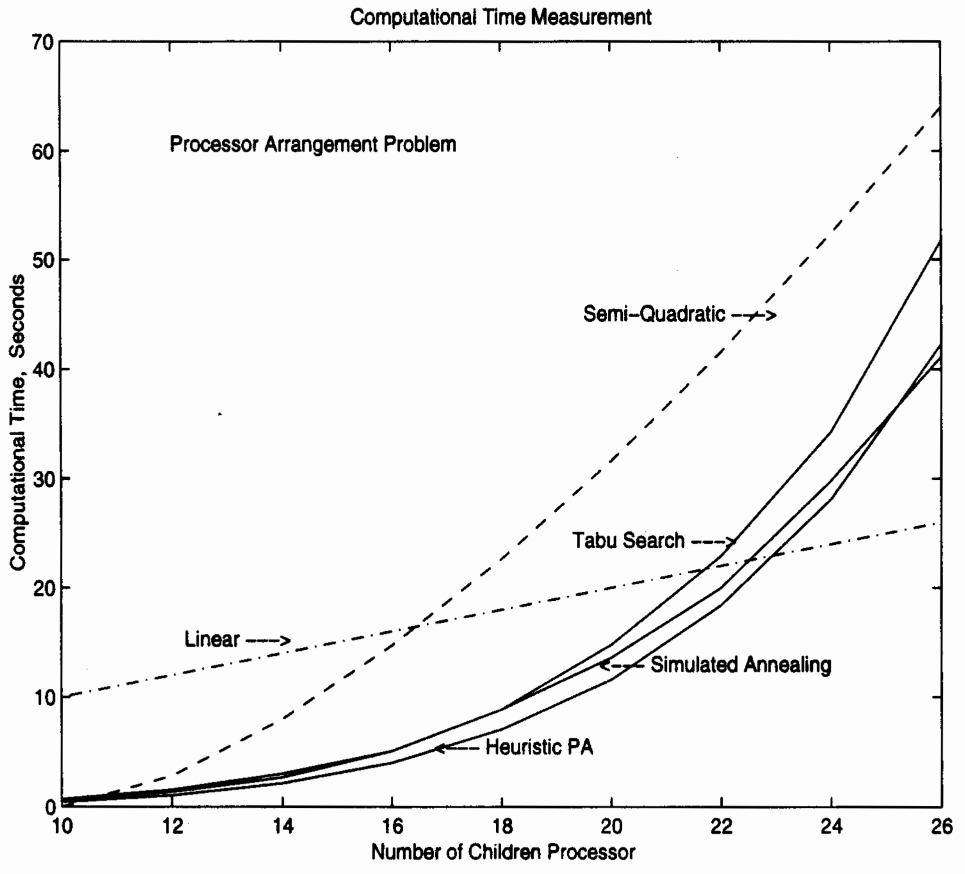

Figure 6: Computational Time of the Processor Arrangement Problem