NORTH-HOLLAND

Informatics and Computer Science

Task Assignment and Transaction Clustering Heuristics for Distributed Systems

JOSE AGUILAR

EHEI, UFR de Mathematiques et Informatique, Université René Descartes, 45 rue des Saints-Peres, 75006 Paris, France

and

EROL GELENBE

Department of Electrical Engineering, Duke University, Durham, North Carolina 27708-0291

ABSTRACT

In this paper, we present and discuss the task assignment problem for distributed systems. We also show how this problem is very similar to that of clustering transactions for load balancing purposes and for their efficient execution in a distributed environment. The formalization of these problems in terms of a graph-theoretic representation of a distributed program, or of a set of related transactions, is given. The cost function which needs to be minimized by an assignment of tasks to processors or of transactions to clusters is detailed, and we survey related work, as well as work on the dynamic load balancing problem. Since the task assignment problem is NP-hard, we present three novel heuristic algorithms that we have tested for solving it and compare them to the well-known greedy heuristic. These novel heuristics use neural networks, genetic algorithms, and simulated annealing. Both the resulting performance and the computational cost for these algorithms are evaluated on a large number of randomly generated program graphs of different sizes.

1. INTRODUCTION

The problem of assigning each task in a parallel program to some processing unit of the system has major impact on the resulting performance. The problem arises in all areas of parallel and distributed computation, where programs are decomposed into tasks or processes, which must then be assigned to processing units for execution.

INFORMATION SCIENCES **97**, 199–219 (1997)

655 Avenue of the Americas, New York, NY 10010

0020-0255/97/$17.00
PII S0020-0255(96)00178-8

In certain systems, this assignment is carried out dynamically at run-time; this gives rise to the *load balancing problem* [26]. However, in many cases, the user or the system will wish to exert explicit control over the assignment of each task. This paper addresses the latter, which is known as the *task assignment problem.*

This problem is very similar to that of clustering transactions for load balancing purposes and for their efficient execution in a distributed environment. These problems can be formalized in terms of a graph-theoretic representation of a distributed program, or of a set of related transactions. The issue is then to minimize an adequate and meaningful cost function by an assignment of tasks to processors or of transactions to clusters.

The task assignment problem is usually addressed using a graph-theoretic representation of the program. Typically, a distributed program is represented as a collection of tasks, which correspond to nodes in a graph. The arcs of the graph may represent communication between tasks, or precedence relations, or both. Task assignment is then formulated as a problem of partitioning the graph so as to minimize some cost function. Typically, each element (or block) in the partition will represent a set of tasks which will be assigned to the same processor. The cost function may represent a combination of communication costs (which will increase as tasks are dispersed among a larger number of processing units) and computation times (which will typically decrease as the number of tasks included in any block becomes smaller). The assignment is then chosen to minimize this combined cost.

In the general case, since this problem is NP-hard, approximate heuristics are needed because exact solutions would require excessive execution times when the number of tasks in the program and the number of processing units are large.

In the following sections, we first introduce task assignment and briefly discuss the related issue of task scheduling. We also discuss load balancing in order to differentiate it with the work presented here. Then, we formalize the task assignment problem and its related cost functions in a graph-theoretic framework. Finally, we survey other work, and present our own approaches and heuristic algorithms to solve the task assignment problem.

The approaches we propose and evaluate in this paper are a heuristic based on the random neural model of Gelenbe [24, 25], a heuristic based on genetic algorithms [5, 27, 53], and the well-known simulated annealing heuristic [2, 5].

2. PROBLEM DEFINITION

Task assignment is simply the choice of a mapping of a set of tasks to a set of processors so as to achieve a predefined goal. This goal is usually represented as some cost function which may consider a combination of several criteria: equitable load sharing between the processors, maximization of the degree of parallelism, minimization of the amount (and delay) of communication between the processors, minimization of the execution time of the program, etc. In order to be of use in achieving a satisfactory solution, the cost function must obviously include the constraints and characteristics of the programs involved (such as task execution times, amount of intertask communication, precedence between tasks), and of the system architecture, including the nature and topology of interconnects between processing units, the speed of the processors, memory system properties (shared or private to processors, limits in memory size, etc.).

Usually, the task assignment problem will not consider the actual schedule or order in which the tasks are executed. On the other hand, task scheduling has been actively researched over the years and precisely addresses this specific issue [16, 29, 44, 50]. Thus in the present paper, we will not discuss scheduling issues.

The related dynamic load balancing, or dynamic task assignment, problem will allocate tasks during program execution [8, 34, 36, 38, 56] and use task migration to shift the workload in the system among processing units [7, 10, 19, 45, 52]. Dynamic load balancing algorithms use system-state information, and the workload may migrate from one processor to another during run-time. *Task migration* is a mechanism where a process on one machine is moved to another machine, that is, it consists in interrupting the task executions and in transferring a sufficient amount of information so that the task can be executed in another place.

Policies for dynamic load balancing, or dynamic task assignment, will often use the following types of rules [10, 19, 45]:

- the *information rule*, which describes how to collect and where to store the information used in making decisions;
- the *transfer rule*, which is used to determine when to initiate an attempt to transfer a task and whether or not to transfer a task;
- the *location rule*, which chooses the machine to or from which tasks will be transferred;
- the *selection rule*, which is used to select a task for transfer.

Dynamic task assignment is obviously better suited to a processing environment which changes frequently due to variations in workload, or due to unexpected events such as processor slowdowns which may occur when a local load has higher priority, processor or network failures, processor withdrawal when a processor is preempted by a higher-priority job, etc. Dynamic load balancing is itself quite complex and the redistribution process creates additional overhead that can adversely impact system performance. Krueger and Livny [35] show that while initial task assignment is capable of improving performance, the addition of task reallocation mechanisms, in many cases, can provide considerable additional improvement.

In the sequel, we will only deal with the task assignment problem.

2.1. GRAPH-THEORETIC APPROACH TO TASK ASSIGNMENT

The graph-theoretic approach to task assignment models a program as a graph, and then uses graph-theoretic techniques to solve the problem. Each task in the parallel program is modeled as a node in the graph, and communicating tasks are connected by an edge. Both nodes and arcs will, in general, be weighted, the first representing task execution times, while the latter represent communication times or amounts of data being exchanged. Both directed and nondirected graphs may be used to represent programs. A nondirected graph will only represent information exchange and concurrent execution between tasks, while a directed graph will deal also with precedence relations between tasks.

There is a very substantial literature about the graph-theoretic approach to task assignment. Stone et al. [51] use the "max flow-minimal cut" theorem of Ford and Fulkerson [21] to search for an optimal assignment which will minimize the communication cost of a system with two processors. In [40], an extension of this approach to a system with n processors is proposed, by recursively using the same theorem combined with a greedy algorithm to find suboptimal assignments. The approach is augmented to include the interference cost, which reflects the degree of incompatibility between tasks. In [41], the same author describes two heuristic algorithms to find suboptimal assignments of tasks to processors. Both algorithms model the task assignment problem as a graph-partitioning problem and show that an appropriate goal is the minimization of the total interprocessor communication cost while meeting a constraint on the number of tasks assigned to each processor. Shen et al. [49] present a graph-matching approach using the minimax criterion, based on both minimization of interprocessor communication and balance of processor

loading. Task assignment is transformed into a type of graph matching, called weak homomorphism. The search of the optimal weak homomorphism corresponds to the optimal task assignment. Both [12] and [54] propose to use the critical-path notion to assign tasks to processors so as to minimize program execution time based on task graph precedence. Ercal et al. [20] present a recursive algorithm based on the Kernighan-Lin bisection heuristic for the effective mapping of the tasks of a parallel program onto a hypercube parallel architecture. Heuristic algorithms are potentially fast, though some (such as those which are based on simulated annealing) can be rather slow. However, they are not guaranteed to yield an optimal solution.

2.2. HEURISTIC ALGORITHMS FOR TASK ASSIGNMENT

Much work on approximate algorithms has been devoted to the minimization of the communication time resulting from task assignment [18, 22], while other work has also included the minimization of execution time [49, 14]. Other research also considers the effect of precedence constraints [15]. Task assignment in real-time systems [47] requires that deadlines be taken into consideration; in [28], the execution time and the degree of parallelism are also optimized while also considering finite memory capacity and task delay. In [13], both load balancing and processor capacity constraints are discussed in the context of real-time and have as a goal the minimization of the communication cost.

Efe [18] presents an algorithm called "2 module clustering" that finds task pairs to be assigned to the same processor. This procedure is run until all the candidate task pairs are grouped together. The goal is to minimize intergroup communication. An improvement of this approach is proposed in [48]. Chu [14] has proposed a similar approach where the minimization of communication cost and load balancing is accomplished in two phases. Tasks are first fused among a set of processors with a clustering algorithm until the number of processors is equal to the number of groups. Task fusion is realized in such a way that two tasks which communicate with each other are assigned either to the same processor or to neighbor processors. Then, one checks to see whether or not the system satisfies the load balancing constraint; if it does not, then some tasks are assigned from processors exceeding the tolerated level of load, to processors that are below this level, while minimizing the cost of communication. More recently, Chu et al. [15] present a method for optimal task allocation that

considers the effects of precedence, intertask communication, and cumulative execution time of each task to search for a minimum-bottleneck assignment.

Bowen et al. [13] propose and evaluate a hierarchical clustering and allocation algorithm, called *A divisible*, that drastically reduces the interprocess communication cost while respecting lower and upper bounds of utilization of the processors. This algorithm is also well suited to dynamic task assignment. Baxter et al. [9] present an algorithm for static task allocation called LAST (Localized Allocation of Static Tasks), which successively allocates sets of tasks to processors, until a completed mapping is constructed. The next task to be allocated is chosen on the basis of connectivity with the previously allocated tasks, and then assigned to processors based on the speed with which the assignment can be computed. The overall cost of the mapping is the time required for the system to execute all the tasks. In [43], several algorithms are given to relocate processes when the system configuration changes. These algorithms modify the process and processor cluster trees generated during the original allocation, to reflect these changes. Generally, only a small subtree of the process cluster tree will have to be remapped to another small processor cluster subtree. Wells et al. [55] have developed a parallel task allocation methodology for nonbuffered message-passing environments. The algorithm incorporates a set of list-based heuristics (priority list, etc.) and graph-theoretical procedures (graph precedence layering, graph width, minimum cut graph traversal for partitioning, etc.) designed to balance computational load with communication requirements. In [54], Tantawi and Towsley study a distributed system composed of a set of heterogeneous processors and develop a technique for static optimal probabilistic assignment. Other work on the static task assignment problem includes [32, 33, 37].

There has also been substantial work on dynamic load balancing and process migration. For example, [38, 39, 42] propose algorithms which migrate tasks from overloaded to less loaded processors, while in [38], a gradient procedure for moving tasks is examined. In [42], a *drafting* algorithm is proposed, based on the idea that a task only has to communicate with a subset of the other tasks. In [11], a model for dynamic task assignment based on "phases" is suggested, where a phase is a complete period of execution of a task, and the idea is to reconsider assignment for every distinct phase of a task. Krueger and Livny [35] study a variety of algorithms combining a local scheduling policy, such as processor sharing, with a global scheduling policy, such as load balancing, to achieve dynamic task allocation. In [26], an adaptive load balancing algorithm is proposed for both tasks and files. The gradient descent paradigm is used to make

on-line load balancing decisions for tasks, and balancing is based on redistribution of files so as to maintain an equal file load at all nodes. Other work on dynamic task assignment using process migration includes [7, 8, 10, 19, 30, 36, 52, 56].

3. FRAMEWORK FOR THE TWO PROBLEMS OF TASK ASSIGNMENT AND TRANSACTION CLUSTERING

The task assignment problem addresses the clustering of a set of tasks on a set of processors so as to optimize system performance. Therefore, we first describe the formal environment within which this problem is considered.

However, this problem is very similar to another important question in distributed systems: how to cluster a set of transactions so that they will be executed on a set of processors. In fact, one may consider that transactions are simply tasks of a special kind. Thus, we will address here the framework for that problem as well.

3.1. TASK GRAPHS AND TASK ASSIGNMENT

In our study, we consider a distributed system architecture which consists of a collection of K processors with distributed memory, i.e., with sufficient memory at each processor so that any one task can be executed. The processors are fully interconnected via a reliable high-speed network.

A parallel program which will be executed in this environment is represented by a *task graph* [23], which is denoted by

$$\Pi = (N, A, e, C),$$

where $N = \{1, \ldots, n\}$ is the set of n tasks that compose the program, $A = \{a_{ij}\}$ is the incidence matrix which describes the graph, and e, C are the amounts of work related to task execution and to communication between tasks. Thus, e_i defines the amount of work—or code to be executed—in task $i = 1...n$. C_{ij} will denote the amount of information transferred during communication from task i to task j, if $a_{ij} = 1$. Clearly, $a_{ij} = 0$ implies that $C_{ij} = 0$.

Note that this model may describe precedence between tasks if the graph is directed and acyclic, or it may be used to represent a set of tasks which interact via passage of information when the graph is not directed (in which case we will have $a_{ij} = a_{ji}$ for all i, j).

The task assignment problem at hand is that of assigning the n tasks to K processors. This means that we have to find a partition $(\Pi_1,\ldots,\Pi_K)$ of the set of n tasks in a way which optimizes performance, as expressed by criteria such as:

- The communication between different processors of the system must be kept to a minimum.
- The load of the different processors must be balanced.
- The total effective execution time of the parallel program must be minimized.

3.2. *CLUSTERING OF TRANSACTIONS*

Now consider once again the task graph model described above with the following differences in the manner in which it is interpreted. We will consider a *transaction graph*:

$$\tau=(T,P,e,C),$$

where $T=\{1,\ldots,n\}$ is a set of n transactions, $P=\{p_{ij}\}$ is the n-by-n precedence matrix which describes the transaction graph, i.e., the precedence dependencies between transactions, and the n-vector e and the n-by-n matrix C represent, respectively, the amount of work related to transaction execution and the amount of information or data granules shared between transactions. Thus, e_i denotes the amount of work—or code to be executed—associated with transaction $i=1...n$. C_{ij} will denote the number of information granules which are shared by transactions i and j.

Clearly, we will seek an assignment of transactions to processing units so that related transactions are executed on the same processor. Related transactions would be those which share common information or data granules, as well as those which have precedence relations between them. Similarly, we would be seeking to *cluster* transactions so that those which have such affinities are placed in the same cluster.

In the sequel, we shall follow the terminology of task assignment, but will keep in mind that a similar approach can be adopted for transaction clustering and that the same algorithms will apply to both problems.

4. TASK ASSIGNMENT AND THE RELATED COST FUNCTIONS

Contrary to load balancing, task assignment usually refers to decisions which are made before program execution, and which are not changed during program execution [6, 9, 17, 41]. Thus, this approach to distributing processes or tasks to processing units can be applied effectively to programs whose run-time behavior is relatively predictable, since the decision must rely on a priori knowledge of the system and of the programs. There are numerous examples of applications where this approach is useful, including major numerical algorithms such as matrix multiplication or inversion, solution methods for differential systems, as well as large nonnumerical problems such as searching and sorting.

As mentioned previously, task assignment is usually carried out so as to optimize a criterion which describes the "costs" or "benefits."

The assignment itself can be characterized by a set of binary variables $\{X_{ip}\}$, where i ranges over the set of tasks and p ranges over the set of processors. Thus, X_{ip} is the binary variable whose value is 1 if task i is assigned to processor p, and is 0 otherwise. For an assignment to be valid, we must have that each task is assigned to exactly one processor:

$$X_{ip}.X_{iq}=0, \quad \text{for all } i,\ p\neq q, \quad \text{and} \quad \sum_{p} x_{ip}=1 \quad \text{for all } i. \tag{1}$$

Since there are a variety of considerations involved, there are different elements which enter into the cost function. We present the main components of the cost function below [4–6, 9, 11, 40].

The total program execution time. This cost depends on the "size" of the tasks, expressed in size of executed code, or in execution time on some normalized processor, and on the speed of the processors. It can be expressed as

$$C_E=\sum_{p}\sum_{i} e_i U_p X_{ip},$$

where e_i is the size of task i in number of instructions executed, and U_p is the (average) time of execution for one instruction on processor p.

The communication cost. This is often considered to be one of the most important factors which need to be minimized by the task assignment. It depends on the quantity of information to be exchanged between tasks, on the interconnection system topology, and on the speed of the communication links:

$$C_C = \sum_p \sum_{q \neq p} \sum_i \sum_{j \neq i} (CCP_{piqj} + CCP_{qjpi} + C_{ij} D_{pq} + C_{ji} D_{qp}) X_{ip} X_{jq},$$

where CCP_{piqj} is the time necessary for processor p to set up the communication between tasks i and j, if i is on p and j on q; note that bilateral communication is assumed; C_{ij} is the total quantity of information transferred between tasks i and j; and D_{pq} is the average time needed to transfer a unit of information from processor p to processor q, after set-up is complete.

The cost of access to the files. Depending on where a task is situated, the time it will take to access files also varies. Specifically, we assume that some file f will be resident on (or accessible through) processor i, if the binary variable Y_{fp} takes the value 1; otherwise it will be 0. The corresponding cost term becomes

$$C_F = \sum_p \sum_{q \neq p} \sum_i \sum_f (CCF_{piqf} + CCF_{qfpi} + V_{if} D_{pq}) X_{ip} Y_{fq},$$

where V_{if} is the average quantity of information that task i needs from file f, and CCF_{piqf} (CCF_{qfpi}) is the average time necessary for processor p to set up communication between task i on processor q and file f on q.

The interference cost. This is meant to represent the fact that when several tasks are assigned to the same processor, specific overhead may be incurred due to competition both for the processor's attention and for the processor's communication subsystem's attention. This overhead is composed of: (i) the competition for the use of the processor's computation (I^e); (ii) the competition to use the communication services of the processor (I^c):

$$C_I = \sum_p \sum_i \sum_{j \neq i} I_{pij} X_{ip} X_{jp},$$

where I_{pij} is the interference cost between the tasks i and j for the processor p:

$$I_{pij} = I^e_{pij} + I^c_{pij},$$

and

$$I^e_{pij} = (e_i + e_j)U_p,$$

$$I^c_{pij} = \sum_{q \neq p} \sum_{t \neq i \delta t \neq j} (CCP_{piqt} + C_{it}D_{pq} + CCP_{pjqt} + C_{it}D_{pq})X_{tq}.$$

The cost of load imbalance. An optimal assignment must assure an equitable distribution of the workload between the processors. A possible form for this cost term is

$$C_B = \frac{1}{K}\left(\sum_p \left(\sum_i e_i U_p X_{ip} - \frac{C_E}{K}\right)^2\right),$$

where C_E is the total (sequential) execution time of the task graph (or the total work it contains), and K is the number of processors in the system.

5. THE HEURISTIC ALGORITHMS USED IN THIS STUDY

Let us now turn to the three heuristic methods we have applied and compared for solving the task assignment problem, namely, a neural network approach, a genetic algorithm approach, and simulated annealing. All of these methods are compared below with the standard greedy algorithm heuristic for the task assignment problem.

In the sequel, we describe each of the methods, including the basic principles involved, as well as the specific technique used for the problem at hand. Then we summarize the experimental results which have been obtained.

5.1. THE RANDOM NEURAL NETWORK MODEL

Neural networks have been used over the last several years to obtain heuristic solutions to hard optimization problems [46].

The random neural network model has been developed by Gelenbe [24, 25] to represent a dynamic behavior inspired by natural neural systems.

This model has a remarkable property called "product form" which allows the direct computation of joint probability distributions of the neurons of the network.

The basic descriptor of a neuron in the random network [24, 25] is the probability of excitation of the M neurons, $q(i)$, $i=1,\ldots,M$, which satisfy a set of nonlinear equations:

$$q(i)=\frac{\sum_{j=1}^{n} q(j)r(j)P^{-}(j,i)+\Lambda(i)}{\sum_{j=1}^{n} q(j)r(j)P^{-}(j,i)+\lambda(i)}, \tag{2}$$

where $\Lambda(i)$ is the rate at which *external excitation signals* arrive to the ith neuron, $\lambda(i)$ is the rate at which *external inhibition signals* arrive to the ith neuron, $r(i)$ is the rate at which neuron i fires when it is excited, and $P^{+}(i,j)$ and $P^{-}(i,j)$, respectively, are the probabilities that neuron i (when it is excited) will send an *excitation* or an *inhibition* signal to neuron j.

Notice that this is a "frequency modulated" model, which translates rates of signal emission into excitation probabilities via (2). For instance, $q(j)r(j)P^{+}(j,i)$ denotes the rate at which neuron j excites neuron i. Eq. (2) can also be viewed as a sigmoidal form which treats excitation (in the numerator) asymmetrically with respect to inhibition (in the denominator).

In order to construct a heuristic for the solution of the task assignment problem, we construct a random neural network composed of $M=nK+K$ neurons, where n is the number of tasks and K is the number of processors.

For each (task, processor) pair (i,u), we will have a neuron $\mu(i,u)$ whose role is to *decide* whether task i should be assigned to processor u. We denote by $q(\mu(i,u))$ the probability that $\mu(i,u)$ is excited, and if this probability is close to 1, we will be encouraged to assign i to u.

In order to reduce communication times in the assignment, and encourage the placement on the same processor of tasks which communicate with each other, $\mu(i,u)$ will *excite* any neuron $\mu(j,u)$ if $a_{ij}=1$ or $a_{ji}=1$, and will tend to *inhibit* $\mu(j,v)$ if $u\neq v$. Similarly, $\mu(i,u)$ will *inhibit* $\mu(j,u)$ if $a_{ij}=0$, $a_{ji}=0$.

Neurons $\mu(i,u)$ and $\mu(i,v)$, $u\neq v$, will *strongly inhibit* each other so as to indicate that the same task should not be assigned to different processors.

For each processor u, we will have a neuron $\pi(u)$ whose role is to let us know whether u is heavily loaded with work or not. If u is very heavily loaded, it will attempt to reduce the load on processor u by *inhibiting* neurons $\mu(i,u)$, and it will attempt to increase the load on processors $v\neq u$ by *exciting* neurons $\pi(u)$. In the same way, $\mu(i,u)$ will *excite* neuron $\pi(u)$ to provide information about processor u's load.

The parameters of the random network model expressing these intuitive criteria are chosen as follows:

$\Lambda(\mu(i,u)) = random,$
$\Lambda(\pi(u)) = n/K,$ to express the desirable equal load sharing property,
$\lambda(\mu(i,u)) = 0,$
$\lambda(\pi(u)) = 0,$
$r(\mu(i,u)) = nK,$
$r(\pi(u)) = n + K - 1,$
$r(\mu(i,u))P^+(\mu(i,u),\mu(j,v)) = 1$ if $(a_{ij} = 1$ or $a_{ji} = 1)$ and $u = v,$
0 otherwise.
$r(\mu(i,u))P^-(\mu(i,u),\mu(j,v)) = 1$ if $u \neq v$ and $(a_{ij} = 1$ or $a_{ji} = 1$ or $i = j),$
or if $a_{ij} = 0$ and $a_{ji} = 0,$
0 otherwise.
$r(\mu(i,u))P^+(\mu(i,u),\pi(v)) = 1$ if $u = v,$
0 otherwise.
$r(\pi(u))P^-(\pi(u),\mu(i,u)) = 1$ if $q(\pi(u)) \rightsquigarrow 1,$
0 otherwise.
$r(\pi(u))P^+(\pi(u),\pi(v)) = 1$ if $q(\pi(u)) \rightsquigarrow 1,$
0 otherwise.

The equations for this case are

$$q(\mu(i,u)) = \left\{ \sum_{a_{ij}=1 \text{ or } a_{ji}=1} q(\mu(j,u)) r(\mu(j,u)) P^+(\mu(j,u),\mu(i,u)) \right\}$$

$$\Bigg/ \Bigg\{ r(\mu(i,u)) + \sum_{v \neq u} \sum_{a_{ij}=1 \text{ or } a_{ji}=1 \text{ or } i=j} q(r(\mu(j,v)) \times P^-(\mu(j,v),\mu(i,u)))$$

$$+ \sum_{v} \sum_{\substack{a_{ij}=0 \\ \delta a_{ji}=0}} q(r(\mu(j,v)) P^-(\mu(j,v),\mu(i,u))$$

$$+ q(\pi(u)) r(\pi(u)) P^-(\pi(u),\mu(i,u))) \Bigg\},$$

$$q(\pi(u)) = \frac{\Lambda(\pi(u)) + \sum_{j=1}^{n} q(\mu(j,u)) r(\mu(j,u)) Pp(\mu(j,u),\pi(u)) + \sum_{v=1}^{K} q(\pi(v)) r(\pi(v)) P^+(\pi(v),\pi(u))}{r(\pi(u))}.$$

The results obtained with this approach and other methods are presented and compared in Section 6.

5.2. *SIMULATED ANNEALING*

Simulated annealing (SA) is a well-known method which uses the physical concepts of "temperature and energy" to represent and solve optimization problems using a Monte Carlo simulation. The objective function of the optimization problem is treated as the "energy" of a dynamical system, while temperature is introduced to randomize the search for a solution. The state of the dynamical system being simulated is related to the state of the system being optimized.

This approach has been applied to numerous examples and is known to provide very good approximations to the optimal solution of combinatorial optimization problems. The simulation runs can be very lengthy, however, and the results are sensitive to the "cooling" procedure that is used, i.e., to the manner in which the temperature parameter is progressively reduced. The idea is to start with an initial solution, and then try to improve it through local changes. It is inspired by the analogy of a physical system behavior in the presence of a hot temperature bath.

The procedure is the following: the system is submitted to high temperature and is then slowly cooled through a series of temperature levels. For each level, we search for the system's equilibrium state through elementary transformations which will be accepted if they reduce the system energy $E_{new} < E_{old}$.

As the temperature decreases, smaller energy increments are accepted, and the system eventually settles into a low energy configuration that is very close, if not identical, to the global minimum.

One has to consider the effect of the initial temperature, the cooling rate, and the threshold (Fac-accep) which define the probability that an uphill move of size Δ will be accepted.

As there are many parameters which need to be set in order to obtain the best results, we have studied their influence one at a time, in hope of isolating their effect. In general, we select values that yield the quickest running time without sacrificing the quality of the solutions found. The best values of these parameters differ from application to application, and possibly, from instance to instance.

If the initial temperature is very high, then the execution time of the heuristic becomes very long, and if it is low, then poor results are obtained [2, 5].

The cooling rate defines the procedure to reduce the temperature: a rapid reduction yields a bad local optimum. Slow cooling is also expensive in CPU time. Good results have been obtained when the reduction factor of the temperature is 0.93 between two steps. For low temperature values, we consider that the system has reached a state near the minimum energy (ground state) which corresponds to an optimal solution; consequently, we decrease the temperature slowly (0.965) when the temperature is already low. Several functions have been proposed to determine the probability of acceptance, normally named "heat bath" functions. We use $\exp(-\Delta/T)$, where T is the temperature, because of its simplicity.

There are other parameters which need to be fixed, including the number of iterations to be performed (L), and the number of identical iterations before it is considered that steady-state has been reached for a given temperature level. We have taken the total number of possible elementary transformations to be $n(K-1)$. Practically, when an elementary transformation has been proposed about 100 times at a given temperature, the equilibrium is considered to be reached, when the number of steps that are tested is $100n(K-1)$.

5.3. GENETIC ALGORITHM

This is an optimization method based on the principles of evolution in biology [27].

A genetic algorithm (GA) follows an "intelligent evolution" process for individuals based on the utilization of evolution operators such as mutation, inversion, selection, and crossover [27, 53, 2]. The idea is to find the best local optimum, starting from a set of initial solutions, by applying the evolution operators to successive solutions so as to generate new and better local minima. The procedure evolves until it remains trapped in a local minimum.

The GA applied in our problem follows the following structure. We create a search space of n-vectors, where every vector v corresponds to a possible assignment of the n tasks. Each element $v(i)$ of the vector takes a value from 1 to K, representing the assignment of task i to one of the K processing units.

We use the cost function of the task assignment problem to determine the cost of every individual. We begin with an initial random population of individuals, and choose those individuals with minimal cost for generating new individuals using the genetic operators. We replace the worst individuals of the current solution by the best individuals which are generated, and keep the population constant. The procedure stops if we exceed a given number of generations without finding a better solution.

In this method, several parameters play an important role, including the maximum number of generations (NUMGEN) tried before the procedure is stopped, the probability (PM) of use of the mutation operator after the crossover operator, and the size of the population.

The first parameter determines the speed-up of the algorithm to reach an optimal solution. If the graph is large, a larger NUMGEN may be needed. We have found that a larger PM parameter will provide better results, especially for large graphs when the crossover operator is ineffective, because it will tend to reproduce very similar individuals. A large PM implies a larger execution time. Talbi et al. [53] consider varying PM dynamically in the population. If the size of the population is large, we generally obtain better results, but the execution time is obviously correspondingly large. For small populations, rapid convergence is possible but an optimal or very good solution will seldom be found.

6. PERFORMANCE COMPARISONS

In this section, we summarize the results we have obtained for the three heuristics described above, which we compare with each other and with the well-known Kernighan-Lin graph-partitioning heuristic [31]. Comparisons are carried out for a large number of randomly generated task graphs having a number of nodes which varies widely.

The evaluations are carried out on a large number of randomly generated task graphs having different numbers of nodes n. The task graphs are randomly generated as follows. For a fixed n, and for each node of the graph, we draw at random the number d of neighbors of the node, from a uniform distribution running from 1 to some maximum value D. Each task in the node is assumed to have an execution time of 1, and the time for communicating between tasks is also taken to have unit value.

Each simulation run then corresponds to the execution of one job (i.e., a single task graph) using only one method for the task assignment (RNA, GA, SA) and one set of parameters.

The cost function which the Kernighan-Lin greedy heuristic, the genetic algorithm, and simulated annealing attempt to minimize is a composite function including the load balancing effect C_B and the communication cost C_C. It is given by

$$C = \frac{1}{K}\left(\sum_p \left(\sum_i e_i U_p X_{ip} - \frac{C_E}{K}\right)^2\right) + \sum_p \sum_{q \neq p} \sum_i \sum_{j \neq i} (CCP_{piqj} + CCP_{qjpi} + C_{ij} D_{pq} + C_{ji} D_{qp}) X_{ip} X_{jq},$$

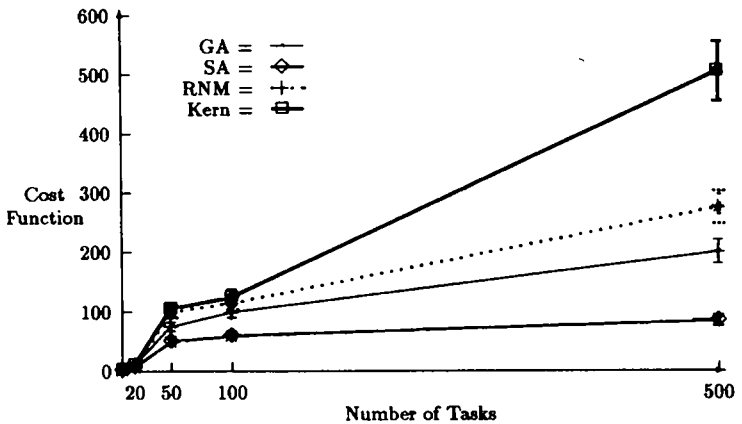

Fig. 1. Resulting cost function C for acyclic graphs.

where we have taken $e_i = 1$, and $C_{ij} = 1$ when $a_{ij} = 1$. Also $D_{pq} = 1$ and the set-up times for communications CCP_{piqj} are zero.

The results we have obtained are summarized in Figures 1 to 4 . These results are obtained for task graphs with $D = 5$, and for both directed acyclic task graphs and for undirected task graphs.

We clearly see that of all the heuristics we have tested, simulated annealing provides the best results with respect to minimizing the cost function. The next best results are obtained using the genetic algorithm.

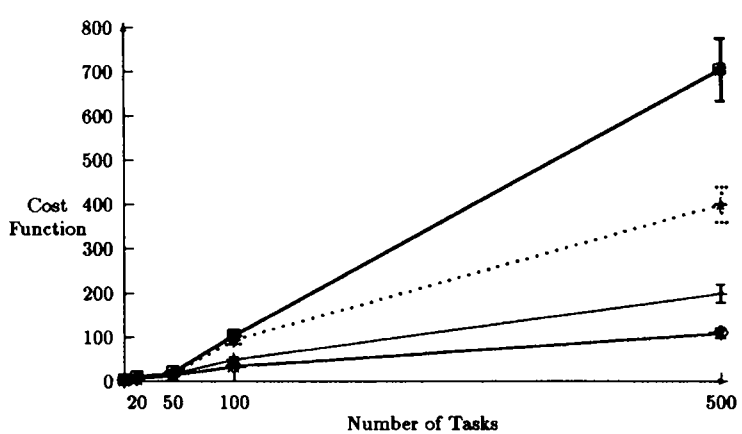

Fig. 2. Resulting cost function C for series-parallel graphs.

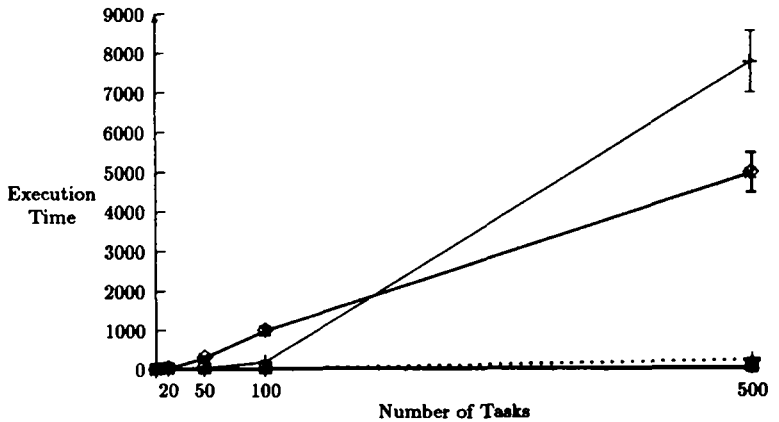

Fig. 3. Execution time of the four heuristics for acyclic graphs.

However, it is also quite clear that these two methods are very time-consuming in program execution time. On the other hand, the Kernighan-Lin heuristic yields the worst results, though it does run very fast. Interestingly enough, the random network model generally provides results which are substantially better than the Kernighan-Lin heuristic, yet substantially worse than either the genetic algorithm or simulated annealing. However, its run-time is comparable to that of the Kernighan-Lin heuristic even for very large task graphs.

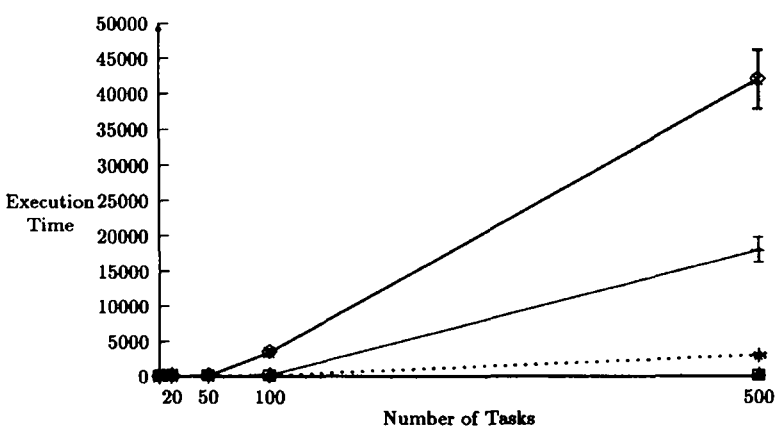

Fig. 4. Execution time of the four heuristics for series-parallel graphs.

The experiments we have run show that the results obtained by each approximate method vary significantly as a function of the size of the graphs considered. However, the relative performance of each of the heuristics tested is consistent over different graph sizes.

Simulated annealing consistently gives the best results, but with a substantially larger execution time than the other approaches. The execution time for the genetic algorithm heuristic is also very large, and can sometimes be larger than that of simulated annealing. This is because the computations for each generation are time-consuming.

The genetic algorithm and the random neural network-based heuristics could be easy to implement on a parallel machine, and this can considerably improve the speed obtained with these methods.

REFERENCES

1. J. Aguilar, Comparison between the random neural network model and other optimization combinatorial methods for large acyclic graph partitioning problem, in: *Proceedings of the 7th International Symposium on Computer and Information Sciences, ISCIS VII*, Antalya, Turkey, 1992.
2. J. Aguilar, Combinatorial optimization methods: A study of graph partitioning problem, in: *Proceedings of the Panamerican Workshop on Applied and Computational Mathematics, PWACM*, Caracas, Venezuela, 1993.
3. J. Aguilar, Heuristic algorithms for task assignment of parallel programs, in: *Proceedings of the International Conference on Massively Parallel Processing, Applications and Development*, Delft, Holland, 1994.
4. J. Aguilar, Resolution du problème de placement de tâches avec de techniques d'optimisation combinatoires, in: *Proceedings 6éme Rencontres Francophones du parallelisme*, Lyon, France, 1994.
5. J. Aguilar, L'Allocation de tâches, l'équilibrage de charge et l'optimisation combinatoire, Ph.D. Thesis, René Descartes University, Paris, France, 1995.
6. F. Andre and J. Pazat, Le placement de tâches sur une architecture parallèle, *Tech. Sci. Inf.* 7:385–401 (1988).
7. S. Baker and K. Milner, A process migration harness for dynamic load balancing, in: *Proceedings of the 14th Technical Meeting of the World Occam and Transputer User Group*, 1991.
8. A. Barak and A. Shilob, A distributed load balancing policy for a multicomputer, *Software Pract. Exp.* 15:901–913 (1985).
9. J. Baxter and J. Patel, The LAST algorithm: A heuristic based static allocation algorithm, in: *Proceedings of the International Conference on Parallel Processing*, Pennsylvania, 1989.
10. G. Bernard, D. Steve, and M. Simatic, Placement et migration dans les systèmes repartis faiblement couplés, *Tech. Sci. Inf.* 10:307–337 (1989).
11. S. Bokhari, On the mapping problem, *IEEE Trans. Computers* C-30:207–214 (1981).
12. S. Bokhari, *Assignment Problems in Parallel and Distributed Computing*, Kluwer Academic, Boston, MA, 1987.

13. N. Bowen, C. Nikolaou, and A. Ghafoor, On the assignment problem of arbitrary process systems to heterogeneous distributed computer systems, *IEEE Trans. Computers* 41:257–273 (1992).
14. W. Chu, L. Holloway, M. Lan, and K. Efe, Task allocation in distributed data processing, *Computers*, pp. 57–68 (1980).
15. W. Chu and M. Lan, Task allocation and precedence relations for distributed real time systems, *IEEE Trans. Computers* C-36:667–679 (1987).
16. J. Colin, Problèmes d'ordonnancement avec délais de communication: Complexité et algorithmes, Ph.D. Thesis, Paris VI University, Paris, France, 1989.
17. D. Du, Allocation de tâches dans les systèmes reconfigurables de type statique, Ph.D. Thesis, Orsay University, France, 1992.
18. K. Efe, Heuristic models of task assignment scheduling in distributed systems, *Computers* 15:50–56 (1982).
19. M. Eskicioglu, Process migration in distributed systems: A comparative survey, Technical Report, University of Alberta, 1990.
20. F. Ercal, J. Ramanujam, and P. Sadayappan, Task allocation onto a hypercube by recursive mincut bipartitioning, *J. Paral. Distrib. Comput.* 10:35–44 (1990).
21. L. Ford and D. Fulkerson, *Flows in Networks*, Princeton University, Princeton, NJ, 1962.
22. A. Gabrielian and D. Tyler, Optimal object allocation in distributed computer systems, in: *Proceedings of the 4th International Conference on Distributed Computing Systems*, Cambridge, MA, 1984.
23. E. Gelenbe, *Multiprocessor Performance*, Wiley, New York and Chichester, 1989.
24. E. Gelenbe, Random neural networks with positive and negative signals and product form solution, *Neural Comput.* 1(4):502–511 (1989).
25. E. Gelenbe, Stable random neural networks, *Neural Comput.* 2(2):239–247 (1990).
26. E. Gelenbe and R. Kushwaha, Incremental dynamic load balancing in distributed system, in: *Proceedings of the International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Durham, NC, 1994.
27. D. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, Reading, MA, 1989.
28. E. Houstis, Module allocation of real-time applications to distributed systems, *IEEE Trans. Software Engrg.* 16:699–709 (1990).
29. J. Hwang, Y. Chow, F. Anger, and C. Lee, Scheduling precedence graphs in systems with interprocessor communication times, *SIAM J. Comput.* 18:244–257 (1989).
30. W. Joosen and P. Verbaeten, On the use of process migration in distributed systems, *Microproc. Microprogr.* 28:49–52 (1990).
31. B. Kernighan and S. Lin, An efficient algorithm for partitioning graphs, *Bell Syst. Tech. J.* (1970).
32. C. Kim and H. Kameda, An algorithm for optimal static load balancing in distributed computer systems, *IEEE Trans. Computers* 41:381–384 (1992).
33. S. Kim and J. Browne, A general approach to mapping of parallel computations upon multiprocessors architectures, in: *Proceedings of the International Conference on Parallel Processing*, Pennsylvania, 1988.
34. O. Kremin and J. Kramer, Methodical analysis of adaptive load sharing algorithms, *IEEE Trans. Paral. Distrib. Syst.* 3:747–759 (1992).
35. P. Krueger and M. Livny, The diverse objectives of distributed scheduling policies, in: *Proceedings of the 7th International Conference on Distributed Computing Systems*, Berlin, Germany, 1987.

36. H. Kuchen and A. Wagener, Comparison of dynamic load balancing strategies, *J. Paral. Distrib. Proc.* 303–314 (1991).
37. K. Kyrimis, Placement of processes and files in distributed systems, Technical Report, Princeton University, 1990.
38. H. Lin and M. Keller, The gradient model load balancing policies, *IEEE Trans. Software Engrg.* SE-13:32–38 (1987).
39. H. Lin and C. Raghavendra, A dynamic load-balancing policy with a central job dispatcher, *IEEE Trans. Software Engrg.* 18:148–157 (1992).
40. V. Lo, Heuristic algorithms for task assignment in distributed systems, *IEEE Trans. Computers* 37:1384–1397 (1988).
41. V. Lo, Algorithms for static task assignment and symmetric contraction in distributed computer systems, Technical Report, University of Oregon, 1988.
42. L. Ni, C. Xu, and T. Gendreau, A distributed drafting algorithm for load balancing, *IEEE Trans. Software Engrg.* SE-11:1153–1159 (1985).
43. C. Nikolaou, D. Ferguson, G. Leitner, and G. Kar, Allocation and relocation of processes in a distributed computer system, *Curr. Adv. Distrib. Comput. Commun.* 1 (1986).
44. C. Papadimitriou and J. Tsitsiklis, On stochastic scheduling with in-tree precedence constraint, *SIAM J. Comput.* 16(1):1–6 (1987).
45. K. Park, Process migration policies in distributed operating systems, *Trans. Inf. Proc.* 31:1080–1090 (1990).
46. G. Peretto, Neural networks and combinatorial problems, in: *Proceedings of the International Conference on Neural Networks*, Paris, France, 1990.
47. K. Ramamritham, J. Stankovic, and W. Zhao, Distributed scheduling of tasks with deadlines and resource requirements, *IEEE Trans. Computers* 38:1110–1123 (1989).
48. M. Schaar, K. Efe, L. Delcambre, and L. Bhuyan, Load balancing with network cooperation, in: *Proceedings of the 11th Conference on Distributed Computing Systems*, Arlington, VA, 1991.
49. C. Shen and W. Tsai, A graph matching approach to optimal task assignment in distributed computing systems using minimax criterion, *IEEE Trans. Computers* C-34:197–203 (1985).
50. N. Shirazi and M. Wang, Analysis and evaluation of heuristic methods for static task scheduling, *J. Paral. Distrib. Comput.* 10:222–232 (1990).
51. H. Stone, Multiprocessor scheduling with the aid of network flow algorithms, *IEEE Trans. Software Engrg.* SE-3:85–93 (1977).
52. J. Soh and V. Thomas, Process migration for load balancing in distributed systems, in: *Proceedings of TENCON'87*, 1987, pp. 888–892.
53. E. Talbi and P. Bessiere, Un algorithme génétique massivement parallèle pour le problème de partitionement de graphes, Rapport de Recherche, Laboratoire de Gènie Informatique, Grenoble, France, 1991.
54. A. Tantawi and D. Towsley, Optimal static load balancing in distributed computer systems, *J. ACM* 32:445–465 (1985).
55. B. Wells, D. Jackson, and C. Caroll, A parallel task allocation methodology for nonbuffered message-passing environments, Technical Report, University of Alabama, 1989.
56. C. Xu and F. Lau, Analysis of the generalized dimension exchange method for dynamic load balancing, *J. Paral. Distrib. Comput.* 16:386–393 (1992).

Received 1 October 1995; revised 1 October 1996