Assigning Task Modules to Processors in a Distributed System

L. Tao[1], B. Narahari[2], Y.C. Zhao[1]

Abstract

This paper studies the problem of allocating interacting program modules, of a distributed program, to the heterogeneous processors in a distributed computer system. The interacting program/task modules are represented by an undirected task graph, whose vertices denote task modules and edges denote interactions between modules. We are given the execution cost of a task module on each processor, the communication cost between two task modules if they are placed on different processors, and the interference cost between two task modules if they are placed on the same processor. The objective of our problem is to assign task modules to the processors such that the total of the above three costs incurred by the program on the system is minimized. The above task assignment problem is known to be NP-hard for a three processor system, but its complexity for a two processor system remained open. In this paper we prove that the problem remains NP-hard for a two processor system even when (1) task graph is planar and has maximum degree 3 or (2) task graph is bipartite. We then present three heuristics, based on simulated annealing, tabu search, and stochastic probe approaches respectively. We present an experimental analysis of these three heuristics, and compare their performance with the only known heuristic method in the literature. Our experiments demonstrate that our heuristics provide major improvements.

[1]L. Tao and Y.C. Zhao are with Dept. of Computer Science, Concordia University, Montreal, Quebec, Canada.

[2]B. Narahari is with Dept. of Electrical Engineering and Computer Science, George Washington University, Washington, DC, USA.

[3]This work has been partially supported by Canadian NSERC research grant OGP0041648, Quebec FCAR research grant 92NC0026, and USA National Science Foundation Grant NSF-CCR-9209345.

JCMCC 14 (1993), pp. 97-135

1 Introduction

The advances in semiconductor technology and computer networks have led to the design and development of a number of distributed computer systems [13,14,17]. A distributed computer system consists of a set of heterogeneous processors, with different computational capabilities, that communicate with each other through an interconnection network. The time, or cost, required by a pair of processors to communicate is determined by the topology of the interconnection network. Distributed systems provide an environment where different modules, or procedures, of a program can be executed on different processors; a module running on one processor can transfer control, using the network, to a module running on a different processor [1].

A program to be executed on a distributed system consists of a set of interacting program modules, henceforth also called task modules. Each program module performs a set of operations, and thus can be viewed as a procedure. To execute the program, we must assign each task module to a processor in the system; a task module is executed on the processor to which it is assigned. Completion of the distributed program is accomplished when all the task modules have been executed on the processors. The performance of the program on the distributed system depends heavily on the following three costs (times):

- *Execution costs.* Since the processors are heterogeneous, different modules usually need different execution times on different processors. For example, a module consisting of floating point operations will have lower execution time (cost) on a processor that has floating point capability, as opposed to a processor without this capability. Thus, the execution time of the program depends on how the modules are assigned to the processors.

- *Communication costs.* If two interacting modules are assigned to different processors, the inter-module communications need to go through the interconnection network thus may introduce significant performance overhead due to the limited interprocessor communication bandwidth. Since different pairs of

modules usually have different communication loads, and different pairs of processors usually have different communication links and bandwidths between them, the communication costs are in general dependent on the assignment of the task modules to the processors.

- *Interference costs.* If two modules are assigned to the same processor, they will compete for the same set of resources (CPU, memory, I/O, etc.) and this interference cost (time) slows down the execution of either module. Since different modules may need different hardware resources, and the processors are heterogeneous, the interference costs are also processor-dependent in general.

A fundamental issue in maximizing the system performance is therefore to assign the modules to the processors to minimize the *total cost* incurred by the program if the program has a sequential control thread; or to minimize the *completion cost,* which is the maximum cost incurred by the program on any processor, if the modules can be executed concurrently. This problem, of finding the assignment of modules to processors, is called the task assignment problem in distributed systems.

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

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

$$\text{cost}_1(\pi) = \sum_{u \in V} X(u, \pi(u)) + \sum_{u \neq v} C(\pi(u), \pi(v), u, v)$$

or the *completion cost*

$$\text{cost}_2(\pi) = \max_{1 \leq k \leq m} \left\{ \sum_{\pi(u)=k} X(u, k) + \sum_{\substack{\pi(u)=k \\ u \neq v}} C(\pi(u), \pi(v), u, v) \right\}.$$

We call the above task assignment problem with objective function $\text{cost}_1(\pi)$ the *nonuniform task assignment* problem (TA). If both the communication costs and the interference costs are processor-independent, then TA is reduced to the *uniform task assignment* problem. While the uniform model is used by most work on task assignment in the literature, it is not a realistic model for heterogeneous computer systems. In such systems different processor pairs may have different communication bandwidths, as is the case in most distributed systems [13]. In this paper we focus on the study of nonuniform task assignment problem, and treat the uniform task assignment as a special case. Some researchers have also worked on uniform task assignment to minimize the completion cost $\text{cost}_2(\pi)$ [15]. In our experimental study, $\text{cost}_2(\pi)$ is provided for reference only.

Most related work in the literature focuses on uniform task assignment problems [16,12,3]. Stone proposed the classical model for uniform task assignment without interference costs and used the max-flow algorithm to derive optimal solutions to minimize the total cost for systems with two processors [16]. Stone also gave important hints for generalizing his algorithm to solve task assignment problems with $m > 2$ [16]. Gursky proved that stone's task assignment model for $m > 2$ is NP-hard [9]. Lo designed a heuristic to address the uniform task assignment problem based on the max-flow algorithm to minimize the total cost for systems with $m \geq 2$, and introduced the interference costs to reduce the completion cost [12].

For the nonuniform task assignment problem, polynomial time algorithms have been developed, for arbitrary m, for special instances of the task graph G. Specifically, there is an $O(m^2n)$ algorithm if G is a tree [2], an $O(m^3n)$ algorithm if G is a series parallel graph [19], or an $O(m^{k+1}n)$ algorithm if G is a partial k-tree [6]. In addition, Fernández-Baca recently proved that the nonuniform task assignment problem is NP-hard for $m \geq 3$ even if G is planar and bipartite [6]. It was also shown that no polynomial time approximation algorithms exist unless P=NP [6], for $m \geq 3$. The complexity of TA for a two processor system remained open. In addition, for the nonuniform task assignment (TA), there have been no heuristics proposed thus far.

In the first part of this paper, we show the intractable nature of the nonuniform task assignment problem for a two processor system. We first show that the problem remains NP-hard, for a two processor system, even if the task graph G is planar with maximum degree 3. We then observe that the problem remains NP-hard for a two processor system even if G is bipartite. Finally, we show that the problem can be solved in polynomial time for arbitrary number of processors if the degree of the graph is at most 2. This final observation closes the gap between the P and NP cases in terms of the degree of the graph.

In the second part of this paper, three heuristics are proposed to solve this problem based on simulated annealing, tabu search, and stochastic probe approaches. An extensive experimental study is performed on 42 different data sets both for nonuniform and uniform task assignment. Experiments show that our heuristics are efficient and effective in general. For uniform task assignment problem, our heuristics on the average improve the total costs of Lo's max-flow based heuristic by 10% for the data sets with constant or zero interference costs (using only 2.8% CPU time of Lo's heuristic); and by 71% for the data sets with random interference costs.

In the next section, we first present our complexity study for the TA problem. Section 3 addresses some problem-specific design issues for general heuristics for task assignment. Three efficient heuristics based on simulated annealing, tabu search, and stochastic probe are proposed in Section 4. Extensive experimental studies, mainly between our heuristics and Lo's max-flow based heuristic, are reported

in Section 5. Section 6 summarize our results with some observations. We summarize the main ideas and drawbacks of Lo's heuristic in the Appendix to this paper.

2 Complexity Study

In this section we prove the NP-hardness of nonuniform task assignment with $m = 2$ by reducing the instances of two standard NP-complete problems to the instances of task assignment with two processors. To facilitate our complexity study, we first reformulate the nonuniform task assignment problem so that a problem instance can be completely described by a graph G_1, and any feasible solution to this problem instance can be represented by a subgraph of G_1.

2.1 Problem reformulation

As an instance of the task assignment problem, we are given the task graph $G = (V, E)$, the cost functions X and C, the number of modules n, and the number of processors m. To facilitate the following complexity analysis, we represent all of the above related information in a weighted *expanded graph* $G_1 = (V_1, E_1)$. We define V_1 as

$$V_1 = \cup_{v \in V} \mathcal{V}_v, \text{ where } \mathcal{V}_v = \{v^1, v^2, \ldots, v^m\}.$$

For any $v \in V$ and $1 \leq k \leq m$, v^k represents an allocation of module v to processor k. We define E_1 as

$$E_1 = \{(u^i, v^j) | (u, v) \in E \text{ and } 1 \leq i, j \leq m\}.$$

The execution and generic costs are represented by the weights of the vertices and edges in G_1 respectively. For each vertex $v^k \in V_1$, v^k is assigned a weight $w_1(v^k) = X(v, k)$. Similarly, for each edge $e = (u^i, v^j) \in E_1$, e is assigned a weight $w_2(e) = C(i, j, u, v)$.

From the definition of the problem, a feasible assignment can now be represented by a subset V_h of V_1 if $| V_h \cap \mathcal{V}_v | = 1$ for all $v \in V$. The *induced subgraph* on V_h, denoted by $H = (V_h, E_h)$ where $E_h = \{(u, v) \in E_1 | u, v \in V_h\}$, is isomorphic to G and represents the execution and generic costs of an assignment denoted by V_h. We call H an *allocation graph.* The cost of an assignment represented by

V_h can now be calculated in terms of the vertex and edge weights of the allocation graph H. Let $W_1(V_h) = \sum\{w_1(v) \mid v \in V_h\}$ and $W_2(E_h) = \sum\{w_2(e) \mid e \in E_h\}$, the cost of the assignment, denoted by $\Gamma(H)$, is:

$$\Gamma(H) = W_1(V_h) + W_2(E_h).$$

The task assignment problem can now be stated as: *Find an allocation graph* $H \subseteq G_1$ *to minimize* $\Gamma(H)$. The decision version of TA can be stated as:

Instance: Expanded graph G_1 and a real number $D > 0$ (cost).
Question: Is there an allocation graph H such that $\Gamma(H) \leq D$.

2.2 Problem Complexity

In this subsection we show the intractable nature of the task assingment problem for a two processor system. Specifically we show that TA remains NP-hard for a two processor system even if the task graph is planar and has maximum degree 3 or the tak graph is bipartite.

Theorem 1 *TA is NP-hard even if $m = 2$ and G is planar with degree 3.*
Proof: It is easy to see that the problem belongs to NP since for any assignment V_h and any cost D we can check in polynomial time $O(|V_h| + |E_h|) = O(|V| + |E|)$ whether $\Gamma(H) \leq D$. To show that it is NP-hard, we show that the decision version of the problem is NP-complete by presenting a polynomial time transformation from any instance of the Planar One-In-3SAT problem, which was shown to be NP-complete in [5], to an instance of the TA problem.

Planar One-In-3SAT Problem

Instance: Given a set $\mathcal{U} = \{u_i \mid 1 \leq i \leq \mu\}$ of variables and a collection $C = \{c_j \mid 1 \leq j \leq \theta\}$ of clauses over $\mathcal{U}$ such that each clause $c_j \in C$ has $|c_j| = 3$ and the graph $G_I = (V_I, E_I)$ is planar where

$$V_I = C \cup \mathcal{U},$$

$$E_I = \{(c_j, u_i) \mid u_i \in c_j \text{ or } \overline{u}_i \in c_j\}.$$

Question: Is there a truth assignment for $\mathcal{U}$ such that each clause in C has exactly one true literal?

Let $\mathcal{U} = \{u_1, u_2, \ldots, u_\mu\}$ and $\mathcal{C} = \{c_1, c_2, \ldots, c_\theta\}$ be an instance of the Planar One-in-3SAT problem, and let G_I be the graphical representation of this instance. Let s_i, for $1 \leq i \leq \mu$, denote the number of clauses in C that contain either u_i or $\bar{u}_i$. We first construct a task graph G as follows. The vertex set of G is

$$V = \{c_j^1, c_j^2, c_j^3 \mid 1 \leq j \leq \theta\} \cup \{u_i^1, u_i^2, \ldots, u_i^{s_i} \mid 1 \leq i \leq \mu\}.$$

The edge set of G is defined such that

$$E = E_C \cup E_{C-U} \cup E_U,$$

where

- $E_C = \{(c_j^1, c_j^2), (c_j^2, c_j^3), (c_j^1, c_j^3) \mid 1 \leq j \leq \theta\}$;
- $E_U = \{(u_i^1, u_i^2), (u_i^2, u_i^3), \ldots, (u_i^{s_i-1}, u_i^{s_i}) \mid 1 \leq i \leq \mu\}$; and
- the edge set E_{C-U} is defined in such a way that $(c_j^a, u_i^b) \in E_{C-U}$ if and only if the a-th literal of clause c_j is either u_i or $\overline{u}_i$, and each vertex u_i^b has exactly one such edge in E_{C-U}.

Since the degree of G is 3 and G_I is planar, we can always make G planar by choosing b for each c_j^a according to the connection order in G_I, first from top to bottom, then from left to right. For example, let $\mathcal{U} = \{u_1, u_2, u_3\}$, $\mathcal{C} = \{c_1, c_2\}$, $c_1 = \{u_1, \bar{u}_2, u_3\}$, and $c_2 = \{\bar{u}_1, u_2, u_3\}$. Figure 1 shows the corresponding G_I. Figure 2 shows the corresponding task graph G.

We now describe a procedure to construct the expanded graph $G_1 = (V_1, E_1)$ of G for a two processor system. We define

$$V_1 = \mathcal{D} \cup \mathcal{F},$$

where $\mathcal{D} = \{c_j^1, \overline{c}_j^1, c_j^2, \overline{c}_j^2, c_j^3, \overline{c}_j^3 \mid 1 \leq j \leq \theta\}$ and $\mathcal{F} = \{u_i^1, \overline{u}_i^1, u_i^2, \overline{u}_i^2, \ldots, u_i^{s_i}, \overline{u}_i^{s_i} \mid 1 \leq i \leq \mu\}$. Each vertex in G is replaced by two vertices where $c_j^a \in V_1$ corresponds to an assignment of module $c_j^a \in V$ to processor 1 and $\overline{c}_j^a \in V_1$ corresponds to an assignment of module $c_j^a \in V$ to processor 2. The vertices in $\mathcal{F}$ are defined similarly. We define

$$E_1 = E_p \cup E_q \cup E_r \cup E_s \cup E_t \cup E_x \cup E_y,$$

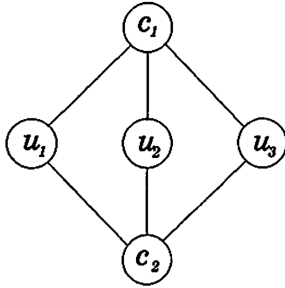


Figure 1: Graph G_I

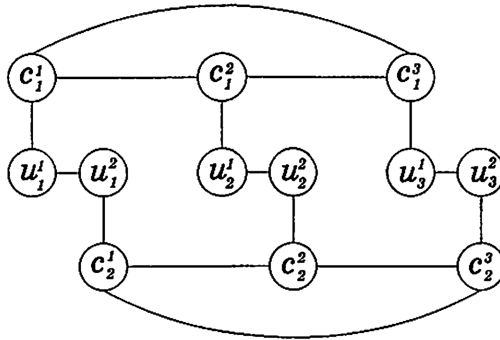


Figure 2: Task graph G

where

- $E_p = \{(c_j^1, c_j^2), (c_j^2, c_j^3), (c_j^1, c_j^3) \mid 1 \leq j \leq \theta\}$;
- $E_q = \{(\overline{c}_j^1, \overline{c}_j^2), (\overline{c}_j^2, \overline{c}_j^3), (\overline{c}_j^1, \overline{c}_j^3) \mid 1 \leq j \leq \theta\}$;
- $E_r = \{(c_j^1, \overline{c}_j^2), (\overline{c}_j^1, c_j^2), (c_j^2, \overline{c}_j^3), (\overline{c}_j^2, c_j^3), (c_j^1, \overline{c}_j^3), (\overline{c}_j^1, c_j^3) \mid 1 \leq j \leq \theta\}$;
- $E_s = \{(u_i^1, u_i^2), (u_i^2, u_i^3), \ldots, (u_i^{s_i-1}, u_i^{s_i}) \mid 1 \leq i \leq \mu\}$;
- $E_t = \{(\overline{u}_i^1, \overline{u}_i^2), (\overline{u}_i^2, \overline{u}_i^3), \ldots, (\overline{u}_i^{s_i-1}, \overline{u}_i^{s_i}) \mid 1 \leq i \leq \mu\}$;
- $E_x = \{(u_i^1, \overline{u}_i^2), (\overline{u}_i^1, u_i^2), (u_i^2, \overline{u}_i^3), \ldots, (u_i^{s_i-1}, \overline{u}_i^{s_i}), (\overline{u}_i^{s_i-1}, u_i^{s_i}) \mid 1 \leq i \leq \mu\}$;
- $E_y = \{(c_j^a, u_i^b), (c_j^a, \overline{u}_i^b), (\overline{c}_j^a, u_i^b), (\overline{c}_j^a, \overline{u}_i^b) \mid (c_j^a, u_i^b) \in E\}$.

To complete the construction of G_1 we need to assign weights to the edges and vertices of G_1. Let $w_1(v) = 1$ for all $v \in V_1$, i.e., all vertices are assigned unit cost. To assign the weight of each edge, let M_1 be any number larger than 12θ and let M_2 be any number larger than $12\theta + 3\theta M_1 - \mu$. The weight of each edge is then defined as follows:

- set $w_2(e) = M_2$ if $e \in E_p \cup E_x$;
- set $w_2(e) = M_1$ if $e \in E_q$;
- set $w_2(e) = 1$ if $e \in E_r \cup E_s \cup E_t$;
- The weights of the edges in E_y are assigned such that if the a-th literal of clause c_j is u_i and $e = (c_j^a, u_i^b) \in E_1$, then set $w_2(e) = 1$ and $w_2(e') = M_1$ for all $e' \in \{(c_j^a, \overline{u}_i^b), (\overline{c}_j^a, u_i^b), (\overline{c}_j^a, \overline{u}_i^b)\}$. If the a-th literal of clause c_j is $\overline{u}_i$ and $e = (c_j^a, \overline{u}_i^b) \in E_1$, then set $w_2(e) = 1$ and $w_2(e') = M_1$ for all $e' \in \{(c_j^a, u_i^b), (\overline{c}_j^a, u_i^b), (\overline{c}_j^a, \overline{u}_i^b)\}$. Figure 3 shows the expanded graph G_1 for the task graph G in Figure 2.

In what follows, we show that there exists a solution for the Planar One-in-3-SAT problem if and only if there exists an allocation graph H such that $\Gamma(H) \leq 3\theta M_1 + 12\theta - \mu$. Figure 4 shows the allocation graph H for our example.

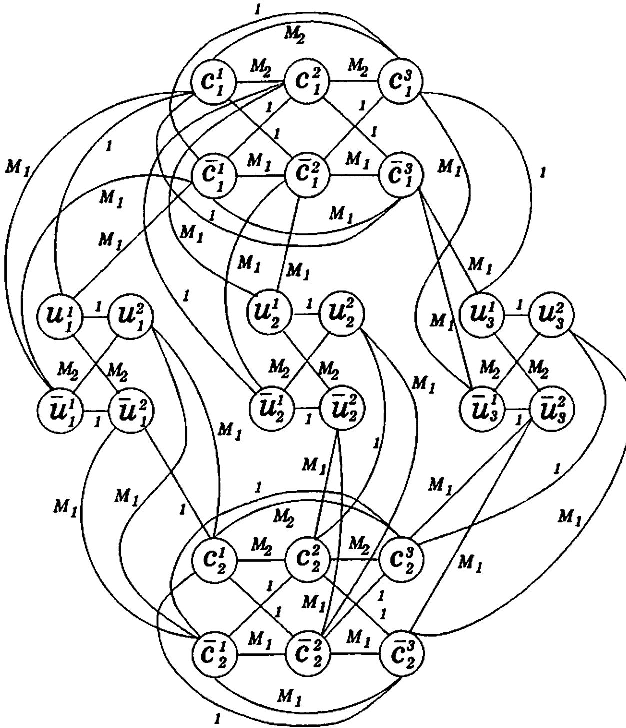
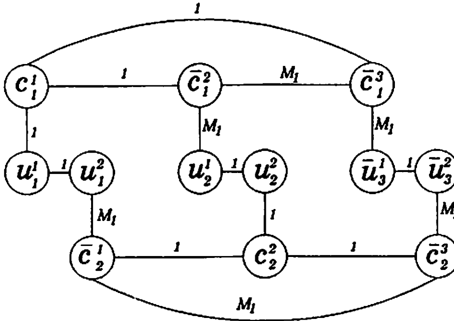


Figure 3: Expanded graph G_1



$$\Gamma = 12 + 2\,(2 + M_1) + 3 + 2\,(2M_1 + 1) = 6M_1 + 21$$

Figure 4: Allocation graph H

Suppose there exists such a truth assignment for U. Using this truth assignment we can construct an allocation graph $H = (V_h, E_h)$ in which

$$V_h = H_1 \cup H_2 \cup H_3$$

where

$$\begin{aligned} H_1 &= \{c_j^a, \overline{c}_j^b, \overline{c}_j^{b'} \mid \text{the } a\text{-th literal of clause } c_j \text{ is true and the } b\text{-th and } b'\text{-th literals of clause } c_j \text{ are false, for } 1 \le j \le \theta\}, \\ H_2 &= \{u_i^1, u_i^2, \ldots, u_i^{s_i} \mid \text{variable } u_i \text{ is true for } 1 \le i \le \mu\}, \\ H_3 &= \{\overline{u}_i^1, \overline{u}_i^2, \ldots, \overline{u}_i^{s_i} \mid \text{variable } u_i \text{ is false for } 1 \le i \le \mu\}; \end{aligned}$$

and

$$E_h = \{(z, w) \in E_1 \mid z, w \in V_h\}.$$

To compute $\Gamma(H)$, it is observed that $\sum\{w_1(v) \mid v \in V_h\} = 6\theta$, since $|V_h| = 6\theta$ and $w_1(v) = 1$ for all V_h; $\sum\{w_2(e) \mid e \in E_h \cap (E_q \cup E_r)\} = \theta(M_1 + 2)$; $\sum\{w_2(e) \mid e \in E_h \cap (E_s \cup E_t)\} = \sum_{i=1}^{\mu}(s_i - 1)$; and $\sum\{w_2(e) \mid e \in E_h \cap E_y\} = \theta(1 + 2M_1)$. Therefore, we have $\Gamma(H) = 6\theta + \theta(M_1 + 2) + \sum_{i=1}^{\mu}(s_i - 1) + \theta(1 + 2M_1)$. Since each clause has 3 literals, thus $\sum_{i=1}^{\mu} s_i = 3\theta$, the above discussion implies that $\Gamma(H) = 3\theta M_1 + 12\theta - \mu$.

Conversely, assume that there exists an allocation graph H of G_1 such that $\Gamma(H) \le 3\theta M_1 + 12\theta - \mu$. Since $M_2 > 3\theta M_1 + 12\theta - \mu$, the weight of any edge e in E_h must be less than M_2, *i.e.*, equal to 1 or M_1. Therefore, for each i, $1 \le i \le \mu$, either $\{u_i^1, u_i^2, \ldots, u_i^{s_i}\} \subseteq V_h$ or $\{\overline{u}_i^1, \overline{u}_i^2, \ldots \overline{u}_i^{s_i}\} \subseteq V_h$, but if $u_i^l \in V_h$ ($\overline{u}_i^l \in V_h$), then $\overline{u}_i^l \notin V_h$ ($u_i^l \notin V_h$). From similar reasoning, for each j, $1 \le j \le \theta$, at most one vertex from $\{c_j^1, c_j^2, c_j^3\}$ can be in V_h, and thus $\sum\{w_2(e) \mid e = (z, w) \in E_h$ such that $z, w \in \mathcal{D}\} \ge (2 + M_1)\theta$. Further, it is noted that $\sum\{w_2(e) \mid e = (z, w) \in E_h$ such that $z \in \mathcal{D}$ and $w \in \mathcal{F}\} \ge (1 + 2M_1)\theta$. The above discussion implies that $\Gamma(H)$ must be at least $\sum\{w_1(v) \mid v \in V_h\} + \sum_{i=1}^{\mu}(s_i - 1) + (2 + M_1)\theta + (1 + 2M_1)\theta$, which is equal to $3\theta M_1 + 12\theta - \mu$. That is, the equality must hold and the only possible case to satisfy the equality is to have exactly one vertex from $\{c_j^1, c_j^2, c_j^3\}$ in V_h for each j, $1 \le j \le \theta$, and the weight of the edge from the vertex in $\{c_j^1, c_j^2, c_j^3\} \cap V_h$ to the vertex in $\mathcal{F}$ is equal to 1. This establishes the existence of a truth assignment for $\mathcal{U}$

such that each clause in C has exactly one true literal. In particular, we can assign the true value to u_i if $\{u_i^1, u_i^2, \ldots, u_i^{s_i}\} \subseteq V_h$, or to $\overline{u}_i$ if $\{\overline{u}_i^1, \overline{u}_i^2, \ldots, \overline{u}_i^{s_i}\} \subseteq V_h$. This completes the proof of the theorem.

The next theorem shows the intractability of the problem, for $m = 2$ even if the task graph is bipartite.

Theorem 2 *TA is NP-hard even if $m = 2$ and G is bipartite.*

Proof: The proof uses a transformation from an instance of the Maximum 2-Satisfiability problem [7] (Max 2-SAT) to an instance of TA.

Let $\mathcal{U} = \{u_1, u_2, ..., u_\mu\}$ be the set of variables, $\mathcal{C} = \{c_1, c_2, \ldots, c_\theta\}$ be the set of clauses and k be the given constant as defined in the Maximum 2-SAT problem. We first construct a bipartite task graph G as follows. The vertex set of G is

$$V = \{c_i | 1 \leq i \leq \theta\} \cup \{u_j | 1 \leq j \leq \mu\}.$$

The edge set of G is:

$$E = \{(c_i, u_j) | \text{ literal } u_j \text{ or } \bar{u}_j \text{ appears in clause } c_i\}.$$

From the construction above it follows that G is bipartite. The expanded graph G_1 for a two processor system is constructed as follows. Assume without loss of generality that the literals in each clause are ordered. The vertex set is defined as:

$$V_1 = \{c_i^1, c_i^2 | 1 \leq i \leq \theta\} \cup \{u_j, \bar{u}_j | 1 \leq j \leq \mu\}.$$

Note that c_i^1 corresponds to assigning module c_i to processor 1 and c_i^2 corresponds to assigning module c_i to processor 2. The edge set of G_1 is defined as:

$$E_1 = \{(c_i^1, u_j), (c_i^1, \bar{u}_j), (c_i^2, u_j), (c_i^2, \bar{u}_j) | \text{literal } u_j \text{ or } \bar{u}_j \text{ appears in clause } c_i \}.$$

The weights of all vertices in G_1 is set to 0, *i.e.*, for all $x \in V_1$ $w_1(v) = 0$. Let $e = (c_i^a, b) \in E_1$, where $a \in \{1, 2\}$ and $b \in \{u_j, \bar{u}_j | 1 \leq j \leq \mu\}$. We set $w_2(e) = 1$ if the a-th literal of clause c_i is b, and $w_2(e) = \theta^2$ otherwise. The task graph G and the expanded graph G_1 thus form an instance of TA where G is bipartite and $m = 2$. We now

prove that there exists a truth assignment for $\mathcal{U}$ that simultaneously satisfies at least k of the clauses in C if and only if there exists an allocation graph $H = (V_h, E_h)$ with $\Gamma(H) \leq k - k\theta^2 + 2\theta^3$.

Suppose there exists a truth assignment for $\mathcal{U}$ that simultaneously satisfies k_0 ($\geq k$) of the clauses in C. One can construct an allocation graph $H = (V_h, E_h)$ as follows. The vertex set of H is:

$$\begin{aligned} V_h &= \{u_i | u_i \text{ is true }\} \cup \{\bar{u}_i | \bar{u}_i \text{ is true }\} \\ &\quad \cup \{c_i^a | \ a \in \{1,2\},\ a=1 \text{ if only if the first literal of } c_i \text{ is true}\}. \end{aligned}$$

To determine the value of $\Gamma(H)$, we recall that only edges have non-zero weights. We note that the number of edges in E_h incident to $c_i^a \in V_h$, for $a = 1$ or 2, is two. Suppose c_i is true and c_i^1 (respectively, c_i^2) is in V_h. Then, the first (respectively, the second) literal of c_i, say z^1 (respectively, z^2), is also in V_h and the weight of edge (c_i^1, z^1) (respectively, edge (c_i^2, z^2)) is equal to one. Since the weight of the other edge incident to c_i^1 or c_i^2 is equal to θ^2, the sum of the weights of these two edges incident to c_i^1 or c_i^2 is equal to $1+\theta^2$. On the other hand, assume that c_i is not true. Then c_i^2 is in V_h and the weights of both edges incident to c_i^2 are $2\theta^2$. This implies that $\Gamma_2(E_h) = k_0(1+\theta^2) + (\theta - k_0)2\theta^2$, which is $k_0(1-\theta^2) + 2\theta^3$. Then $\Gamma(H) = W_2(E_h) \leq k(1-\theta^2) + \theta^3$, because $k_0 \geq k$ and $1 - \theta^2 \leq 0$.

To prove the converse, let $H = (V_h, E_h)$ be an allocation graph with $\Gamma(H) = W_2(E_h) \leq k - k\theta^2 + 2\theta^3$. Then, there are at least k vertices in $V_h \cap \{c_i^1, c_i^2 | 1 \leq i \leq \theta\}$ such that the sum of the weights of the edges incident to those vertices are $1 + \theta^2$. To verify this, suppose that there are only k' ($< k$) such vertices. It follows that since $|V_h \cap \{c_i^1, c_i^2 | 1 \leq i \leq \theta\}| = \theta$, $W_2(E_h) = k'(1+\theta^2) + (\theta - k')2\theta^2$, which is larger than $(k - k\theta^2 + 2\theta^3)$ assuming without loss of generality that $\theta > 1$. These observations establish the existence of a truth assignment for $\mathcal{U}$ that simultaneously satisfies at least k of the clauses in C. In particular, assigning the true value to literal u_i (or $\bar{u}_i$) if $u_i \in V_h$ (respectively, if $\bar{u}_i \in V_h$) gives a desired assignment.

2.3 A tractable subset of nonuniform TA

In this subsection we address the case in which the degree of G is less than three and show that TA can be solved in polynomial time for this case. This result together with the NP-hardness result

completely closes the gap between the polynomially solvable cases and the NP-hard cases in terms of the maximum degree of the task graph G.

Theorem 3 *Given an instance of TA with the degree of G less than 3, TA can be solved in $O(m^3n)$ time for arbitrary m.*

Proof: Note that if the degree of G is less than 3, the graph is either a simple path or a cycle. If the graph is a cycle, we can duplicate a particular vertex to create a simple path. Let v_s and v_t be the two end vertices in G. If the original task graph is a cycle, v_s and v_t are two occurrences of a single vertex.

The corresponding expanded graph G_1 has $O(m^2n)$ edges. If we designate a direction for each edge from the incident vertex closer to some v_s^i, $1 \leq i \leq m$, to the another incident vertex, G_1 becomes a directed acyclic graph. Designating any vertex v_s^i, $1 \leq i \leq m$, in G_1, as the source, we have algorithm [4] to find the shortest path between the source and the rest of the vertices in time $O(m^2n)$.

If the original task graph is a cycle, for each $1 \leq i \leq m$, we invoke the shortest path algorithm to find the shortest path from v_s^i to v_t^i. If the original task graph is a simple path, for each $1 \leq i \leq m$, we invoke the shortest path algorithm to find the shortest path from v_s^i to v_t^j where $1 \leq j \leq m$. The shortest path among the resulting m paths represents the optimal task assignment. The time complexity of this process is therefore $O(m^3n)$.

3 Problem-specific design issues

Since uniform and nonuniform task assignment problems are both NP-hard, heuristic approaches must be adopted for their general solutions. In the rest part of this paper, we propose three iterative heuristics to solve the task assignment problems. All these heuristics are based on general solution space search strategies. In this section, we first address some design issues specific to the task assignment, which are common to all the three heuristics, so that our heuristics can be described more concisely. Without loss of generality, we assume that the cost function $C(i, j, u, v)$ is symmetric in terms of both parameter pair (u, v) and parameter pair (i, j): for any integers $i, j \in [m]$ and any modules $u, v \in V$,

$$C(i,j,u,v) = C(i,j,v,u) = C(j,i,u,v) = C(j,i,v,u).$$

3.1 Move and neighborhood design

Let X be the set of all mappings $V \rightarrow \{1,2,\ldots,m\}$. We call X the *solution space.* The nonuniform task assignment problem can be presented as

$$\text{Minimize} \quad \text{cost}_1(\pi): \quad \pi \in X$$

where $\text{cost}_1(\pi)$ is the objective function.

A wide range of heuristics solving problems capable of being written in this form can be characterized conveniently by reference to sequences of *moves* that lead from one trial solution (selected $\pi \in X$) to another. Let S be the set of all defined moves. We use $S(\pi)$ ($\pi \in X$) to denote the subset of moves in S applicable to π. For any $s \in S(\pi)$, $s(\pi)$, the new solution obtained by applying move s to π, is called a *neighbor* of π. We call $\{s(\pi)|s \in S\}$ the *neighborhood* of solution π in solution space X.

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

Our experiments show that module moves in S_1 are very effective in distributing the modules among the processors to minimize the total execution cost, while module swaps in S_2 are very effective in refining the assignment to minimize the total communication and interference costs. The best order and mixture of module moves and module swaps are problem instance dependent. To compromise the neighborhood size and the effectiveness of the moves, our heuristics use a special set S_3 of moves, where for any $\pi \in X$, $S_3(\pi) = S_1(\pi) \cup S_2'(\pi)$, and

$$\begin{aligned} S_2' = \{ \text{ exchange } u \text{ and } v \mid u,v \in V; & \text{ moving } v \text{ to processor } j \\ & \text{ maximizes gain which is } < 0; \\ & u \text{ is assigned to processor } j\}, \end{aligned}$$

where *gain*, the improvement in cost, is defined in the next subsection. Informally, we give module moves higher priority than module swaps. For a given assignment π and a given module v assigned to processor i, we first try moving v to all the other processors; if moving v to processor j has the best gain which is less than zero, then we also try swaps of v with each of the modules assigned to processor j. Experiments show that S_3 performs better than S_1 or S_2 alone in terms of both running time and solution quality for all of our three task assignment heuristics.

3.2 Gain functions and their incremental update

During each iteration of our heuristics, we need to measure the effectiveness of each potential move. For any move s, we define the gain of s relative to the current assignment $\pi \in X$ to be $g(s) = \text{cost}_1(\pi) - \text{cost}_1(s(\pi))$. For each $s = (u,i) \in S_1$, we have the gain of s relative to the current solution $\pi \in X$ to be

$$\begin{aligned} g_1(u,i) &= X(u,\pi(u)) - X(u,i) \\ &\quad + \sum_{\substack{x \in V \\ x \neq u}} \{C(\pi(u),\pi(x),u,x) - C(i,\pi(x),u,x)\}. \end{aligned}$$

For each $s = (u,v) \in S_2$, we have the gain of s relative to the current solution $\pi \in X$ to be

$$\begin{aligned} g_2(u,v) &= g_1(u,\pi(v)) + g_1(v,\pi(u)) \\ &\quad -2C(\pi(u),\pi(v),u,v) \\ &\quad +C(\pi(u),\pi(u),u,v) \\ &\quad +C(\pi(v),\pi(v),u,v). \end{aligned}$$

Since the gains for moves in S_3 are defined in terms of g_1 or g_2, and g_2 can be easily obtained from g_1, we only need to maintain the current definition of g_1. If we store g_1 in the memory, after a module move or swap, g_1 can be incrementally updated efficiently. After a move $s = (u,i) \in S_1$, the function g_1 can be updated as follows: let

$$\begin{aligned} a &= X(u,i) - X(u,\pi(u)) \\ &\quad + \sum_{\substack{x \in V \\ x \neq u}} \{C(i,\pi(x),u,x) - C(\pi(u),\pi(x),u,x)\}; \end{aligned}$$

for any $t \in [m]$, let

$$g_1'(u,t) = g_1(u,t) + a;$$

and for any $x \in V - \{u\}$ and any $t = [m] - \{\pi(x)\}$, let

$$\begin{aligned} g_1'(x,t) = \; & g_1(x,t) \\ & -C(\pi(x), \pi(u), x, u) \\ & +C(\pi(x), i, x, u) \\ & +C(t, \pi(u), x, u) \\ & -C(t, i, x, u) \end{aligned}$$

where g_1' denotes the updated version of g_1.

4 Iterative heuristics for task assignment

Simulated annealing [11,10] and tabu search [8] are two of the most important techniques for general combinatorial optimization. Even though they are new (having a history less than 10 years) and still under development, they have claimed success in many application domains. Stochastic probe is a new approach recently proposed by us for combinatorial optimization [18]. It combines the advantages of both the stochastic search process in simulated annealing and the aggressive search process in tabu search. In this section we summarize the main ideas of these three approaches, and present our adaptations of them to the task assignment problem.

4.1 Simulated annealing

Simulated annealing can be viewed as an enhanced version of the local search. It attempts to avoid entrapment in poor local optima by allowing occasional uphill moves. This is done under the influence of a random number generator and a control parameter called the *temperature*. As typically implemented [11], the simulated annealing approach involves a pair of nested loops and two additional parameters, a *cooling ratio* r, $0 < r < 1$; and an integer *temperature length* L (see the generic simulated annealing heuristic in Figure 5).

1. Get a random initial solution π.
2. Get an initial temperature $T > 0$.
3. While stop criterion not met do:
    3.1 Perform the following loop L times:
        3.1.1 Let π' be a random neighbor of π.
        3.1.2 Let $\Delta = \text{cost}_1(\pi) - \text{cost}_1(\pi')$.
        3.1.3 If $\Delta \geq 0$ (downhill move), set $\pi = \pi'$.
        3.1.4 If $\Delta < 0$ (uphill move), set $\pi = \pi'$ with probability $e^{\Delta/T}$.
    3.2 Set $T = rT$ (reduce temperature).
4. Return the best π visited.

Figure 5: Simulated annealing

The heart of this procedure is the loop at Step 3.1. Note that $e^{\Delta/T}$ will be a number in the interval $(0, 1)$ when $T > 0$ and $\Delta < 0$, and rightfully can be interpreted as a probability that depends on Δ and T. The probability that an uphill move will be accepted diminishes as the temperature declines, and, for a fixed temperature T, small uphill moves have higher probabilities of acceptance than larger ones. This particular method of operation is motivated by a physical analogy, best described in terms of the physics of crystal growth [11]. It has been proven that the heuristic will converge to a global optimum if the temperature is lowered exponentially and the initial temperature is chosen sufficiently high [10].

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

As for the cooling schedule design, we made the following deci-

sions.

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

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

3. For each temperature we measure the acceptance rate of the proposed moves. The heuristic stops when for five temperatures the acceptance rate is lower than *MINPERCENT* and the best visited solution is not improved in that period of time. Here MINPERCENT is another parameter in the range $(0, 1)$.

All the parameters for our simulated annealing heuristic are not independent. We tune the parameters of our annealing heuristic for each of our problem instance one at a time. We repeat the process until no perturbation of the parameters can improve the performance. We find that the parameter values $r = 0.95$, SIZEFACTOR $= 16$, INIPROB $= 0.4$, and MINPERCENT $= 0.02$ are appropriate for most of the problem instances.

4.2 Tabu search

Tabu search is another newer general approach for combinatorial optimization [8]. It differs from simulated annealing at two main aspects:

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

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

Figure 6 outlines a generic tabu search heuristic using π to represent a solution, $\text{cost}_1(\pi)$ the cost function, and t the length of the tabu list. Given a random solution, the heuristic repeats the loop at

1. Get a random initial solution π.
2. While stop criterion not met do:
   2.1 Let π' be a neighbor of π maximizing $\Delta = \text{cost}_1(\pi) - \text{cost}_1(\pi')$ and not visited in the last t iterations.
   2.2 Set $\pi = \pi'$.
3. Return the best π visited.

Figure 6: Tabu search

Step 2 until some stop criterion is met. During each iteration, the heuristic makes an exhaustive search of the solutions in the neighborhood of the current solution which have not been traversed in the last t ($t > 1$) iterations. The neighboring solution with the best cost will be used to replace the current solution. The main design issues for a tabu search heuristic are as follows:

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

2. The design of the contents of the tabu list. If move s is used to transform the current solution to π, the corresponding cell of the tabu list should capture some attributes of π or s so that π will not be traversed again in the next t steps. At one extreme, we can store solution π directly in the tabu list. But in practice, to save memory space and checking time, some attributes of s will be stored in the tabu list to prevent s or s^{-1} to be used in the next t iterations. If we use a more detailed set of attributes of a solution or move in each cell of the tabu list, more memory space and checking time will be incurred during the solution-space search, and the searches will be less restrictive since less solutions (in addition to the ones visited in the last t iterations) will be tabued. On the other hand, if we use a more abstract (simplified) set of attributes of a solution or move in each cell

of the tabu list, the implementation will be more space and time efficient for each iteration, and the searches will be more restrictive since more extra solutions will be tabued.

3. The design of the aspiration level function. To make the implementation more space and time efficient, most designs of the contents of the tabu list will tabu too many solutions in addition to those visited in the last t iterations, thus risk to lose good move candidates. As a make-up, we can define an aspiration level $A(s, \pi)$ (usually an integer) for each pair of move s and solution π such that if $\text{cost}_1(s(\pi)) < A(s, \pi)$ the tabu status of s for the current solution π can be overridden. In practice some attributes of π, instead of π itself, will be used in the definition of $A(s, \pi)$. $A(s, \pi)$ is designed to capture the common properties of the earlier applications of s to solutions sharing the same attribute values as π.

4. The design of the length t of the tabu list. Parameter t determines how long the move history will be saved in the tabu list. Suppose that π is a local optimum, and it needs at least t' consecutive "uphill" moves to go to another local optimum π'. Then $t \geq t'$ is a necessary condition for π to reach π'. In general, the longer the tabu list, the more time for tabu status checking for each move, and the more restrictive the search process. On the other hand, a too short tabu list risks to introduce cycling in the solution space. Parameter t can be a constant or a variable during the execution of the heuristic. For many applications, a tabu list length around 7 is found appropriate [8].

Following is a description of our tabu search heuristic for task assignment.

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

2. For the tabu list design, we use a circular list to maintain the vertices moved (swapped) in the last t ($t > 1$) iterations. We find that a more detailed characterization of the past moves usually traps the search process in a small subspace of the

solution space (many vertices may never be moved). A constant tabu list length of 5 produces the best performance for most of our problem instances.

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

4.3 Stochastic probe

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

- As for the simulated annealing heuristic, it is not aggressive in neighborhood search. Each iteration chooses a random neighboring solution, which is usually not the most profitable one. The solution cost improves mostly in a narrow time range. The solution searches after this range is mainly limited to a small subspace of X [18].
- As for the tabu search heuristic, the utilization of information is low. For example, if we use S_3 to define the moves, then each iteration needs to search a neighborhood of more than $n(m-1)$ solutions while using the information for only few (no more than the length of the tabu list plus one) of the neighboring solutions. The deterministic search process also limits the solution search to a small subspace of X.

The objective of this subsection is to introduce a new approach for general combinatorial optimization. We will demonstrate its power through the task assignment problem in the next section. The design is based on our following convictions:

- Aggressive neighborhood searches are essential to finding "good" solutions in a practical time frame. But a more aggressive search usually implies more search time. While tabu search and simulated annealing approaches represent the two

extremes, a good trade-off must be made to compromise the aggressiveness and the running time of the search process.

- A good search heuristic should have the ability to effectively leave local optima when they are reached. The trace of the current solution should be controlled by the recent move history, not by "random walk."
- Randomized search is more effective in avoiding cycling in solution space than the tabu list technique. But the acceptance of moves with very bad gains (as simulated annealing does in high temperature) is usually not profitable.

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

Given any $\pi \in X$ and $v \in V$, we use $\tilde{S}(\pi, v)$ to denote the subset of moves in $S(\pi)$ that redefines $\pi(v)$. For any integer $p \geq 0$, we use random$(-p)$ to represent a random integer between $-p$ and 0 inclusively. Figure 7 outlines our stochastic probe heuristic. Informally, the heuristic consists of a sequence of well-organized probes, each probe searching for a local optimum. The last solution in a probe will be modified randomly to some extent to become the initial solution for the next probe. The heuristic stops when no improvements on the best visited solution occur for several consecutive probes. Each probe in turn consists of a sequence of iterations, each making an aggressive search for the most profitable move involving the current vertex. All the vertices become the current one in turn, thus no one will be ignored in the search process. Variable p is used to control the tolerance of "bad" moves. The chosen move will be accepted if and only if it has a gain greater than random$(-p)$. Initialized to be p_0, p will increase its value if the last move has negative gain, and reset to p_0 as soon as a move is accepted. This mechanism is designed to help the solution search leave local optima when they are reached. A probe finishes when the gains for the last k consecutive iterations are all less than or equal to zero. The following are the main design issues to apply this approach to the solution of a particular problem.

1. Get a random initial solution π.
2. Initialize p to p_0.
3. Let L be a circular list of the vertices in V.
   Set v to any of the vertices in V (the current vertex).
4. While stop criterion not met do:
   4.1 While there is any $\Delta > 0$ in the last k iterations of this loop do:
      4.1.1 Let v be the next vertex down the list L.
      4.1.2 Let $s \in \tilde{S}(\pi, v)$ maximizing Δ in 3.1.3.
      4.1.3 Let $\pi' = s(\pi)$, $\Delta = \mathrm{cost}_1(\pi) - \mathrm{cost}_1(\pi')$.
      4.1.4 Let x be the average of negative gains encountered in the current execution of loop at Step 3.1.
      Set $p = p - \lfloor \alpha x \rfloor$.
      4.1.5 If $\Delta >$ random$(-p)$, set $\pi = \pi'$, $p = p_0$.
   4.2 Perturb randomly the value of $\pi(u)$ for β% of the vertices u in V.
   4.3 Set $p = p_0$.
5. Return the best π visited.

Figure 7: Stochastic probe

- The parameter p_0. The value of p_0 determines the initial value of p for each probe, which controls the extent of tolerance for "bad" moves.

- The parameter α. α is a real number controlling the sensitivity of the value of p to the recent move history.

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

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

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

- We set p_0 to 20% of the average absolute value of the negative gains for the first 1000 iterations.
- We set α to a value ranging from 0.2 to 0.8 depending on problem instance.
- We set β to a value ranging from 10 to 15 depending on problem instance. We find that the variation of β during the execution cannot significantly improve the solution quality for this particular problem.
- We set k to n. A larger k makes a more thorough probe into a subspace of X with more running time.
- The heuristic stops when the best visited solution cannot be improved for several consecutive probes.

5 Experimental studies

Extensive experiments are conducted to evaluate the relative performances of our heuristics and Lo's heuristic. The experiments can be classified into two parts: (i) experiments based on the nonuniform model; (ii) experiments based on the uniform model. The latter is further subdivided into three categories: (1) experiments without interference costs; (2) experiments with constant interference costs less than the corresponding minimal communication costs; (3) experiments with random interference costs. The data sets are basically generated following Lo's experiment designs in [12]. For each experiment, we report the total cost and CPU time. We also report the completion cost for reference, even though it is not part of the objectives under our models. Let n be the number of task modules, and m the number of processors. The communication pattern of a problem instance can be (a) clustered, in which there are roughly $3m$ clusters; (b) sparse, in which only 1/6 of the elements in C are nonzero; or (c) structured, in which the inter-module communication pattern can be line, ring, square 2-D mesh, or binary tree. For all of our data sets, the costs are randomly generated. The name of each data set begins with "C" for clustered, "S" for sparse, or the name of topology for structured communication patterns, followed by n and m. All of the

measurements	C30_3			C40_4		
	SP	SA	TS	SP	SA	TS
total cost	984	1022	1042	1433	1498	1508
completion cost	205	245	217	318	300	314
CPU time (sec.)	1.07	1.13	1.32	2.5	10.98	8.71

measurements	C50_5			C60_6		
	SP	SA	TS	SP	SA	TS
total cost	2212	2274	2255	2672	2721	2711
completion cost	476	601	585	527	492	491
CPU time (sec.)	3.77	15.55	11.98	9.06	10.7	35.73

Table 1: Performance comparisons for nonuniform clustered data sets

experiments are performed on a SUN Sparc 2 workstation running SUN-OS Release 4.01. The efficient *lift-to-front* max-flow algorithm with time complexity $O(n^3)$ [4] is used to implement Lo's heuristic. To simplify presentation, we use LO, SA, SP, and TS to denote Lo's heuristic and our heuristics based on simulated annealing, stochastic probe, and tabu search respectively.

5.1 Nonuniform task assignment

Since we cannot find general heuristics for this model in the literature, in this subsection we compare the performances of our own three heuristics. We generate 12 problem instances with n ranging from 30 to 60 and m from 3 to 6. All the execution costs, communication costs, and interference costs range from 1 to 10. The intra-cluster costs range from 15 to 50. The experimental results for clustered data sets, sparse data sets, and structured data sets are reported in Tables 1-3. The experimental results show that compared with SA and TS, the average improvements of SP over the 12 problem instances are 3.2% and 1.9% respectively, the CPU times of SA and TS are 1.94 and 2.4 times of those for SP respectively.

5.2 Uniform task assignment

Since LO is the only general heuristic for the uniform model in the literature, in this subsection we mainly compare the performance of our heuristics with that of LO. The experiments for this model

measurements	S30_3			S40_4		
	SP	SA	TS	SP	SA	TS
total cost	856	868	868	1218	1258	1236
completion cost	158	146	146	182	207	222
CPU time (sec.)	1.18	1.23	1.35	1.83	2.93	5.52

measurements	S50_5			S60_6		
	SP	SA	TS	SP	SA	TS
total cost	1687	1726	1690	2208	2216	2217
completion cost	309	271	324	349	350	382
CPU time (sec.)	3.95	6.75	8.87	9.11	23.83	15.45

Table 2: Performance comparisons for nonuniform sparse data sets

measurements	Line60_6			Ring60_6		
	SP	SA	TS	SP	SA	TS
total cost	1421	1439	1424	1400	1402	1425
completion cost	100	134	111	113	105	91
CPU time (sec.)	11.7	17.58	23.28	17.20	23.92	44.68

measurements	Mesh49_6			Tree60_6		
	SP	SA	TS	SP	SA	TS
total cost	1060	1119	1107	1433	1461	1437
completion cost	126	167	134	110	144	101
CPU time (sec.)	9.6	15.5	28.3	19.03	21.25	27.8

Table 3: Performance comparisons for nonuniform structured data sets

measurements	C40_3				C60_4			
	SP	SA	TS	LO	SP	SA	TS	LO
total cost	1699	1705	1701.6	1809	2783.8	2788.2	2785	2962
completion cost	1133.2	1105.2	1128	1608.4	1825	1711.6	1759.2	2866.8
CPU time (sec.)	0.14	0.69	0.6	1.14	0.74	1.58	2.08	19.81

measurements	C80_5			
	SP	SA	TS	LO
total cost	3865.6	3899	3912.8	4431.4
completion cost	2779	2628.8	2787.6	3812
CPU time (sec.)	0.46	1.72	2.73	102

Table 4: Performance comparisons for clustered data sets without interference costs

can be classified into three categories: (a) experiments without interference costs; (b) experiments with constant interference costs; (c) experiments with random interference costs. The execution costs range from 20 to 80 for (a) and (b), and from 1 to 10 for (c). The communication costs range from 5 to 10 for (a) and (b), and from 1 to 10 for (c). The intra-cluster communication costs range from 10 to 25. Ten data sets are used in each category of the experiments. Each data set contains five problem instances generated by the same specifications. The costs reported for each data set are the average over the five problem instances contained in the data set.

5.2.1 Experiments without interference costs

For experiments in this category, no module assignment can be made by Lo's stage 1 (Grab) in 25 of all 50 tested problem instances. For the remaining 25 instances, only 8.7% of the modules are assigned by Lo's stage 1. Tables 4-6 show that compared with LO, the average improvements of SP, SA, and TS over all 50 problem instances are 11.8%, 10.2%, and 10.3% respectively for the total cost. The average CPU times for LO are 77.9, 18.2, and 28.2 times of those for SP, SA, and TS respectively.

measurements	S40_3				S60_4			
	SP	SA	TS	LO	SP	SA	TS	LO
total cost	2027.6	2029.4	2039.6	2052.4	3237.4	3268	3256.2	3502.6
completion cost	1583	1728.4	1704	1956.6	2972	2965	3086.6	3137.4
CPU time (sec.)	0.21	0.9	0.39	3.31	0.12	0.22	0.44	16

measurements	S80_5			
	SP	SA	TS	LO
total cost	4138.6	4280.2	4236.6	4613
completion cost	4089.6	4209.2	4171	4351
CPU time (sec.)	0.36	0.57	0.58	100.57

Table 5: Performance comparisons for sparse data sets without interference costs

measurements	Line64_4				Ring64_4			
	SP	SA	TS	LO	SP	SA	TS	LO
total cost	2484	2484.6	2487	3041	2574.6	2582.2	2577.6	3122.4
completion cost	921	981	916	2216.8	909.8	828.2	919.6	1762.6
CPU time (sec.)	0.38	0.88	2.53	4.94	0.33	1.81	2.52	11.96

measurements	Mesh64_4				Tree63_4			
	SP	SA	TS	LO	SP	SA	TS	LO
total cost	2855.6	2875	2865.8	3261.4	2412	2416.6	2420.8	2908.6
completion cost	1734.2	1606.2	1680.6	3088	983.4	928.4	959.4	2592.8
CPU time (sec.)	0.43	1.74	1.72	3.26	0.42	2.04	2.19	15.19

Table 6: Performance comparisons for structured data sets without interference costs

measurements	C40_3				C60_4			
	SP	SA	TS	LO	SP	SA	TS	LO
total cost	2347.6	2347.6	2348.2	2525	4262	4271.4	4266	4665.2
completion cost	1337.8	1337.8	1294.2	2325	3026.2	2749.4	2848.6	4150.4
CPU time (sec.)	0.28	1.26	0.97	2.5	0.77	1.11	1.31	33.38

measurements	C80_5			
	SP	SA	TS	LO
total cost	6352.2	6386.4	6412	6428
completion cost	5353.2	5596.6	5595.4	6367.2
CPU time (sec.)	0.5	2.12	3.49	75.48

Table 7: Performance comparisons for clustered data sets with constant interference costs

measurements	S40_3				S60_4			
	SP	SA	TS	LO	SP	SA	TS	LO
total cost	2446.2	2489	2484	2645.2	4247	4297	4253.8	4397.4
completion cost	1875.8	1901.4	1965.4	2455	3533	3600.2	4005.8	4150.2
CPU time (sec.)	0.15	0.79	0.53	2.2	0.25	0.59	0.49	18.48

measurements	S80_5			
	SP	SA	TS	LO
total cost	6048	6157.6	6128.6	6640.6
completion cost	5902.6	5786.2	5910.2	6009.8
CPU time (sec.)	0.39	1.17	0.74	90.67

Table 8: Performance comparisons for sparse data sets with constant interference costs

5.2.2 Experiments with constant interference costs

For experiments in this category, all the interference costs are set to constant 4. No module assignment can be made by Lo's stage 1 in 22 of the 50 tested problem instances. For the remaining 28 instances, only 9.1% of the modules are assigned by Lo's stage 1. Tables 7-9 show that compared with LO, the average improvements of SP, SA, and TS over 50 problem instances are 9.87%, 9.42%, and 9.52% respectively for the total cost. The average CPU times over all problem instances for LO are 58, 20, and 22.7 times of those for SP, SA, and TS respectively.

measurements	Line64_4				Ring64_4			
	SP	SA	TS	LO	SP	SA	TS	LO
total cost	2576.2	2578.2	2582.6	3005	2683.2	2686	2687	3276
completion cost	1041.8	1041.8	1028	1415	1051.6	992.6	1011.8	1838.6
CPU time (sec.)	0.49	1.96	2.34	2.7	0.52	1.25	2.44	9.68

measurements	Mesh64_4				Tree63_4			
	SP	SA	TS	LO	SP	SA	TS	LO
total cost	3049.6	3049.8	3051.2	3525.6	2581.4	2584.8	2581.4	3093
completion cost	1611	1617	1583	3304.8	1023.8	1005.8	1013.6	2727.4
CPU time (sec.)	0.73	0.98	3	5.87	0.49	2.35	2.22	11.88

Table 9: Performance comparisons for structured data sets with constant interference costs

measurements	C40_3				C60_4			
	SP	SA	TS	LO	SP	SA	TS	LO
total cost	1852.2	1871.4	1857	4187.8	3522	3536.4	3533.4	9569
completion cost	838.2	897.8	823.4	4173.8	1300.8	1279.8	1339.8	9558.6
CPU time (sec.)	0.22	0.55	1.21	0.02	0.4	1.79	2.86	0.09

measurements	C80_5			
	SP	SA	TS	LO
total cost	5661.6	5714.6	5672	16349.2
completion cost	1754.8	1820.8	1817.8	16067
CPU time (sec.)	0.54	1.23	2.9	0.37

Table 10: Performance comparisons for clustered data sets with random interference costs

5.2.3 Experiments with randomly generated interference costs

For experiments in this category, all the interference costs are generated randomly in the range from 1 to 10. Since all the problem instances have some negative weights in Lo's *reduced network*, only stage 3 (Greedy) of Lo's heuristic can be invoked and the CPU time is thus substantially reduced. Tables 10-12 show that the average total cost over all 50 problem instances for LO are 3.51, 3.47, and 3.49 times of those for SP, SA, and TS respectively. The average CPU times for SP, SA, and TS over all problem instances are 5.36, 15.32, and 23.53 times of those for LO respectively.

measurements	S40_3				S60_4			
	SP	SA	TS	LO	SP	SA	TS	LO
total cost	1670	1696.2	1677.2	4537.8	3093.2	3100.2	3096.4	9892.6
completion cost	704.2	726.4	738.4	4537.8	1140.2	1076	1061.8	9889.4
CPU time (sec.)	0.15	0.64	0.67	0.03	0.52	1.35	2	0.1

measurements	S80_5			
	SP	SA	TS	LO
total cost	4794.4	4809.2	4811.8	17640
completion cost	1430.8	1456.2	1443.6	17639.2
CPU time (sec.)	0.63	2.19	4.4	0.22

Table 11: Performance comparisons for sparse data sets with random interference costs

measurements	Line64_4				Ring64_4			
	SP	SA	TS	LO	SP	SA	TS	LO
total cost	2561.8	2580	2567	11493.8	2533.6	2538.6	2545.4	11357
completion cost	747.4	755.2	754.6	11493.8	710.4	730.6	726.2	11357
CPU time (sec.)	1.09	2.25	2.18	0.14	1.16	2.62	2.65	0.14

measurements	Mesh64_4				Tree63_4			
	SP	SA	TS	LO	SP	SA	TS	LO
total cost	2710.6	2736.8	2732.6	11425.2	2504	2525.2	2516.6	11109.2
completion cost	822.8	807.2	818.6	11424.6	705	722.4	721.4	11108.6
CPU time (sec.)	0.5	1.12	2.43	0.12	0.56	1.66	2.21	0.15

Table 12: Performance comparisons for structured data sets with random interference costs

It can be seen from Tables 4-12 that LO's completion cost and total cost are very close for all the problem instances, since it assigns most of the modules to a single processor.

6 Conclusion

In this paper, we discussed the intractable nature of the task assignment problem in distributed systems. For the two processor system, we first showed that the problem remains NP-hard even if the task graph is planar with maximum degree 3. We then showed that the problem is still NP-hard for the two processor system even if the task graph is bipartite. Finally, we showed that the problem can be solved in polynomial time for the systems with arbitrary number of processors if the degree of the task graph is at most 2. This observation closes the gap between the P and NP cases in terms of the degree of the task graph.

We proposed three efficient and effective heuristics based on simulated annealing, tabu search, and stochastic probe to solve the task assignment problem. An extensive experimental study was performed on different data sets both for nonuniform and uniform task assignment problems. For uniform task assignment problem, our heuristics on the average improve the total costs of Lo's max-flow based heuristic by 10% for the data sets with constant or zero interference costs (using only 2.8% CPU time of Lo's heuristic); and by 71% for the data sets with random interference costs. Among our heuristics, stochastic probe always outperforms simulated annealing and tabu search both in total costs and CPU time for all of the problem instances.

Our heuristics can also be adapted to solve the task assignment problem which minimizes the completion time. A more sophisticated scheme is needed for the incremental update of the current cost to maintain the efficiency of the heuristics. The related results will be reported in another paper.

Appendix: Lo's heuristic

In this appendix we summarize the main ideas and drawbacks of Lo's max-flow based heuristic for task assignment [12]. It will provide the readers with insights to explain why Lo's heuristic performs poorly in general.

Since Lo's heuristic is designed to solve the uniform task assignment problem, we need to redefine some notations. A problem instance for uniform task assignment is defined by n, the number of modules; m, the number of processors; $X(i,j)$, the execution cost of module i on processor j, $1 \leq i \leq n$, $1 \leq j \leq m$; $C(i,j)$, the communication cost between modules i and j, $1 \leq i,j \leq n$, if they are assigned to different processors; and $I(i,j)$, the interference cost between modules i and j, $1 \leq i,j \leq n$, if they are assigned to the same processor.

Allocation for one processor

For any $k \in [m]$, we can allocate task modules to processor k by the max-flow algorithm. Let processor k be the source node, t the sink node representing all the other $m-1$ processors, and each task module a separate node. For any $i \in [n]$, the weight between processor k and module i is

$$w_{ki} = \frac{1}{n-1}\sum_{r \neq k} X(i,r) - \frac{n-2}{n-1}X(i,k) + \frac{1}{2(n-1)}\sum_{j=1}^{n} I(i,j),$$

and the weight between sink t and module i is

$$w_{ti} = \sum_{j \neq k} w_{ji} + \frac{1}{2}\sum_{j=1}^{n} I(i,j) = X(i,k) + \frac{1}{2}\sum_{j=1}^{n} I(i,j).$$

For any $i,j \in [n]$, $i \neq j$, the weight between modules i and j is $C(i,j) - I(i,j)$. All the modules to the source side of a minimal cut are allocated to processor k.

Lo's heuristic

Lo's heuristic has the following three stages:

Grab: Try sequentially to allocate the modules to each processor until no assignment can be made for any of the processors. The execution costs need to be adjusted after each round of allocation. It has been proved that all the assignments made in this stage are the prefix of an optimal assignment [12]. Let T be the set of indices of the unallocated modules after this stage.

Lump: Evaluate a lower bound

$$L = \sum_{i \in T} \min_j X(i,j) + \min_{i \in T - \{i_0\}} \text{cut}(i, i_0)$$

for the cost incurred by allocating the modules indexed by T to at least two processors, where i_0 is any index in T, and $\text{cut}(i, i_0)$ is the minimum cut between modules i and i_0 in the subgraph consisting only of the module nodes indexed by T. If the minimum cost incurred by allocating all of the modules indexed by T to a particular processor is less than or equal to L, allocate accordingly.

Greedy: Evaluate the average weight $\bar{c}$ of $C(i,j) - I(i,j)$ over all pairs of modules i and j. If there is a path between two modules indexed by T consisting only of edges with weight greater than $\bar{c}$, the two modules belong to the same cluster as long as all of the modules in the cluster can be executed on some processor with a limited cost. Each cluster of modules indexed by T is allocated to the same processor incurring minimum execution cost.

Main drawbacks of Lo's heuristic:

1. Max-flow algorithms only work when all the edges have non-negative weights.

   (a) Since the weights for edges between modules are of form $C(i,j) - I(i,j)$, Lo's first two stages can be applied only if $C(i,j) \geq I(i,j)$ for all $i, j \in [n]$. This is a severe restriction for the choice of communication and interference costs.

(b) Even if $C(i,j) \geq I(i,j)$ for all $i,j \in [n]$, the weight w_{ki} between the source and module i will be negative for sufficiently large n and small average interference cost. Stone proposed to add a positive integer Δ to all the weights w_{ui} for $u \in [m]$ to make them nonnegative. Since one Δ will be added to the weight between the source and module i, while $m-1$ Δ's will be added to the weight between module i and the sink t, Stone's approach greatly reduces the chance that module i be allocated.

2. Lower bound L is too conservative. As a consequence, stage 2 is never utilized for all of our experiments. Lo never showed the applicability of stage 2 through any example in her thesis or papers.

3. Stage 3 separates the consideration for communication/interference costs and that for execution costs. Experiments show that stage 3 has the tendency of allocating modules to few processors, thus leading to poor completion costs.

References

[1] S. Bokhari, "Assignment problems in parallel and distributed computing," *Kluwer Academic Publishers,* 1987.

[2] S. Bokhari, "A shortest tree algorithm for optimal assignments across space and time in a distributed processor system," *IEEE Trans. Software Engineering,* SE-7, 1981, pp. 583–589.

[3] W.W. Chu, L. J. Holloway, M. Lan, and K. Efe, "Task allocation in distributed data processing," *Computer,* November 1980, pp. 57–69.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, "Introduction to algorithms," *The MIT Press,* 1990.

[5] M. E. Dyer and A. M. Frieze, "Planar 3DM is NP-Complete," *J. Algorithms,* 7 (1986) pp. 174–184.

[6] D. Fernández-Baca, "Allocating modules to processors in a distributed system," *IEEE Trans. on Software Engineering,* Vol. 15, No. 11, Nov. 1989, pp. 1427–1436.

[7] M. R. Garey and D.S. Johnson, "Computers and intractability: a guide to the theory of NP-completeness," *W.H. Freeman, San Francisco,* 1979.

[8] F. Glover, "Tabu search – Part 1," *ORSA Journal on Computing,* Vol.1, No.3, 1989, pp. 190-206.

[9] M. Gursky, "Some complexity results for a multi-processor scheduling problem," *private communication with H. S. Stone,* 1981.

[10] B. Hajek, "Cooling schedules for optimal annealing," *Mathematics of Operations Research,* Vol.13, No.2, 1988, pp. 311-329.

[11] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon, "Optimization by simulated annealing: an experimental evaluation; Part I, graph partitioning," *Operations Research,* Vol.37, No.6, November-December 1989, pp. 865–892.

[12] V. M. Lo, "Heuristic algorithms for task assignment in distributed systems," *IEEE Trans. on Computers,* Vol. 37, No. 11, November 1988, pp. 1384–1397.

[13] R. M. Metcalfe and D. R. Boggs, "Ethernet: distributed packet switching for local computer networks," *Communications of the ACM,* Vol. 19, July 1976, pp. 395–404.

[14] M. Satyanarayanan, "Multiprocessors: a comparative study," *Prentice-Hall,* Englewood Cliffs, N.J., 1980.

[15] C. C. Shen and W. H. Tsai, "A graph matching approach to optimal task assignment in distributed computing systems using a minmax criterion," *IEEE Trans. on Computers,* Vol. C-34, No. 3, March 1985, pp. 197-203.

[16] H. S. Stone, "Multiprocessor scheduling with the aid of network flow algorithms," *IEEE Trans. on Software Engineering,* Vol. SE-3, No. 1, Jan. 1977, pp. 85–93.

[17] R.J. Swan, S.H. Fuller, and D.P. Siewiorek, "*Cm**– A modular, multi-microprocessor," *Proc. AFIPS 1977 Fall Joint Computer Conference,* No. 46, 1977, pp. 637–644.

[18] L. Tao and Y. C. Zhao, "Multi-way graph partition by stochastic probe," *Computers & Operations Research,* Vol. 20, No. 3, 1993, pp. 321–347.

[19] D. F. Towsley, "Allocating programs containing branches and loops within a multiple processor system," *IEEE Trans. on Software Engineering,* Vol. SE-12, October 1986, pp. 1018–1024.