Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
The Journal of Systems and Software 80 (2007) 724–735
The Journal of Systems and Software
www.elsevier.com/locate/jss

Task allocation for maximizing reliability of a distributed system using hybrid particle swarm optimization

Peng-Yeng Yin *, Shiuh-Sheng Yu, Pei-Pei Wang, Yi-Te Wang

Department of Information Management, National Chi Nan University, 303 University Road, Puli, Nantou 545, Taiwan

Received 28 February 2006; received in revised form 4 August 2006; accepted 5 August 2006
Available online 18 September 2006

Abstract

In a distributed computing system, a number of program modules may need to be allocated to different processors such that the reliability of executing successfully these modules is maximized and the constraints with limited resources are satisfied. The problem of finding an optimal task allocation with maximum system reliability has been shown to be NP-hard; thus, existing approaches to finding exact solutions are limited to the use in problems of small size. This paper presents a hybrid particle swarm optimization (HPSO) algorithm for finding the near-optimal task allocation within reasonable time. The experimental results show that the HPSO is robust against different problem size, task interaction density, and network topology. The proposed method is also more effective and efficient than a genetic algorithm for the test-cases studied. The convergence and the worst-case characteristics of the HPSO are addressed using both theoretical and empirical analysis.
© 2006 Elsevier Inc. All rights reserved.

Keywords: Task allocation problem; Distributed computing systems; Distributed system reliability; Hybrid strategy; Particle swarm optimization; Genetic algorithm

1. Introduction

The configuration of a distributed computing system (DCS) involves a set of cooperating processors communicating over the communication links. A distributed program running in a DCS consists of several modules that need to be allocated to the processors and inter-communicate through the links for the completion of program execution. To improve the performance of a DCS, several issues arise such as the minimization of execution and communication cost (Lee and Shin, 1997; Murthy, 1999; Ernst et al., 2001), the maximization of system reliability and safety (Kartik and Murthy, 1997; Srinivasan and Jha, 1999), and the achievement of better fault tolerance using software and hardware redundancy (Kartik and Murthy, 1995; Hsieh, 2003). Meanwhile, resource constraints may be imposed by memory size of processors and capacity of communication links. This paper investigates the task allocation problem that aims to maximize the system reliability subject to resource constraints.

Distributed system reliability (DSR) has been defined by Kumar et al. (1986) as the probability for the successful completion of distributed programs which requires that all the allocated processors and involved communication links are operational during the execution lifetime. There are two major DSR evaluation approaches in the literature. Kumar et al. (1986) evaluated the distributed program reliability (DPR) by searching all of the minimal file spanning trees (MFST's), which provide accessibility to the required data files for the program. Then the DSR can be computed by multiplying the DPR's of all distributed programs. However, some system parameters such as the execution times of programs and communication loads on the links are not considered in this model. They assume that all processors and communication links have constant reliability. Shatz et al. (1992) proposed another DSR evaluation model where failures from processors or communication

* Corresponding author. Tel.: +886 49 2910960; fax: +886 49 2915205.
E-mail address: pyyin@ncnu.edu.tw (P.-Y. Yin).

0164-1212/$ - see front matter © 2006 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2006.08.005

links are time-dependent, which fits the scenario that modules with longer execution or communication times will increase the failure probability of involved processors or communication links.

Unfortunately, the computational complexity for evaluating the DSR has been shown to be NP-hard (Lin and Chen, 1997). Researchers, however, have developed alternative algorithms for tackling this problem. These methods can be divided into two categories: exact algorithms and approximation algorithms. The *exact algorithms* strive to find an optimal task allocation for small-sized instances. Kartik and Murthy (1997, 1995) used the idea of branch and bound with underestimates and reorder the modules according to module independence for reducing the computations required. Verma and Tamhankar (1997) employed the branch and bound technique for solving the reliability-based multiple join problem in distributed database management systems. The A^* algorithm has also been adopted for finding an optimal task allocation (Shatz et al., 1992). The A^* algorithm first converts the problem to a state-space search tree and trims the tree by a lower-bound estimation function such that the search space is reduced. However, the worst-case complexity of the A^* algorithm is exponential and it is limited to small problems.

The *approximation algorithms*, on the other hand, derive sub-optimal task allocations within reasonable times. Kartik and Murthy (1997, 1995) also developed a heuristic approach from their exact algorithm by assuming that the best solution is more likely to be found in the least cost path thereby reducing the worst-case time complexity of the algorithm. Srinivasan and Jha (1999) proposed a clustering-based heuristic which groups heavily communicating modules into clusters in order to reduce the intermodule communication (IMC) as much as possible. Genetic algorithms (GAs) have also been adopted for solving the problem and obtained promising results. Vidyarthi and Tripathi (2001) used a simple GA to maximize the reliability of DCS with task allocation. Hsieh (2003) and Hsieh and Hsieh (2003) proposed a hybrid GA that combines the GA with a local search procedure. The experimental results show that the hybrid GA produces better task allocation than the simple GA.

GAs (Goldberg, 1989) belong to a branch of computational intelligence called *metaheuristic*. The development of the metaheuristic optimization theory has been flourishing during the last decade (Glover, 1989; Dorigo and Gambardella, 1997; Kennedy and Eberhart, 1995). Applying metaheuristic algorithms for conquering the task allocation problem has several benefits. (1) Exact algorithms search for optimal solutions and are thus computationally intensive, while metaheuristic algorithms deriving near-optimal solutions within reasonable times are more suitable for real-time applications. (2) Many successful applications (Shigenori et al., 2003) have shown the superiority of metaheuristic algorithms over heuristic algorithms in terms of quality of the final solutions obtained, so that careful design and implementation of the metaheuristic algorithms can improve the results substantially.

In this paper, we present a particle swarm optimization (PSO)-based algorithm for solving the task allocation problem with the goal of maximizing the system reliability. A hill-climbing heuristic is embedded in the PSO iteration to expedite the convergence. The experimental results reveal that the proposed hybrid PSO algorithm produces better task allocation than a genetic algorithm on a large set of simulated problem instances, and the difference is larger for large problems.

The remainder of this paper is organized as follows. Section 2 describes the formulation of the task allocation problem for maximizing reliability. Section 3 presents the proposed hybrid PSO algorithm in detail. Section 4 reports the comparative performance and convergence analysis. Finally, Section 5 concludes this work.

2. Problem formulation

2.1. Notation

The notations used in problem formulation are listed in Table 1.

2.2. Problem statement

We consider the task allocation problem with the following scenarios.

- The processors involved in the DCS are heterogeneous. Hence, the processors may be constrained with various units of memory and computation resources and they may have different processing speeds and failure rates. Moreover, the communication links may have different bandwidths and failure rates. A communication

Table 1
Notations used in the problem formulation

Symbol	Description
x_{ik}	Decision variable: $x_{ik} = 1$ if module i is allocated to processor k, and $x_{ik} = 0$ otherwise
n	Number of processors
r	Number of modules
p_k	Processor k
l_{kb}	Communication link connecting p_k and p_b
λ_k	Failure rate of processor p_k
μ_{kb}	Failure rate of communication link l_{kb}
e_{ik}	Incurred accumulative execution time (AET) if module i is executed on processor k
c_{ij}	Incurred intermodule communication (IMC) cost between modules i and j if they are executed on different processors
w_{kb}	Transmission rate of communication link l_{kb}
m_i	Memory resource requirements of module i from its execution processor
M_k	Amount of memory resource capacitated with processor p_k
s_i	Computation resource requirements of module i from its execution processor
S_k	Amount of computation resource capacitated with processor p_k

subsystem is assumed to handle the interprocessor communication, and the communication can be performed concurrently.

- The execution of a module will consume a specific amount of memory and computation resource from its assigned processor. Two modules, if executed on different processors, may communicate with each other and incur a specific amount of intermodule communication (IMC) cost measured in some unit of data quantity.
- A module may take different accumulative execution time (AET) if it is executed on different processors. An amount of IMC cost may take different durations of transmission time if transmitted through different communication links.
- The state of processors and communication links is either operational or down. Failure events are statistically independent.

The above assumptions are basically similar to those presented by Shatz et al. (1992) from which several task allocation techniques with reliability maximization have been developed (Kartik and Murthy, 1997; Srinivasan and Jha, 1999; Kartik and Murthy, 1995; Hsieh, 2003; Verma and Tamhankar, 1997; Vidyarthi and Tripathi, 2001; Hsieh and Hsieh, 2003). We do not consider the precedence constraints among task modules since the component that fails during the idle period can be replaced by a spare and will not affect system reliability. More deliberated considerations towards a realistic task-scheduling model can be found in Sinnen et al. (2006). They combine the contention-aware scheduling and the involvement scheduling where the former takes into account all resource and precedence constraints and the latter accounts for the execution time incurred by the processor involvement for preparing and during the transfer of the communication.

Our problem is to search for an optimal task allocation that maximizes the DSR and satisfies all of the resource constraints. A task execution process in a DCS can be described by the *processor interaction graph* (PIG) and the *task interaction graph* (TIG). The PIG illustrates how the processors are connected in the network topology of the computation environment. Fig. 1 gives some typical examples of PIG, including linear, ladder, cross, tree, and star, where the nodes indicate the processors and the edges correspond to the communication links. The TIG renders the intermodule communication cost incurred by the mission. Fig. 2 shows an example of a TIG where the intermodule communication costs c_{ij} among five modules are specified. An important characteristic of TIG is the *task interaction density*, denoted by d, which measures how communication intensive a task is. We define d as the ratio of the number of intermodule communication requests to the number of pairs of different modules. As d increases, the intermodule communication becomes more intensive and the reliability derived could be lower due to involvement of more communication links. Moreover, the CPU time required will slightly increase with large d because of the extra computations for the reliability related to those involved communication links.

To evaluate the DSR for the successful completion of a mission with a task allocation, we follow Shatz's formulation (Shatz et al., 1992) to compute the probability that all involved components (processors and communication links) are operational during the mission. Under a task allocation $\boldsymbol{X} = \{x_{ik}\}_{1\leqslant i\leqslant r, 1\leqslant k\leqslant n}$, the reliability of processor p_k during a time interval t follows the Poisson distribution, $R_k(\boldsymbol{X}) = \mathrm{e}^{-\int_0^t \lambda_k(t)\mathrm{d}t}$, and it reduces to $\mathrm{e}^{-\lambda_k t}$ if we assume that the processor failure rate λ_k is constant during the mission. Since the total elapsed time for executing the modules assigned to p_k is $\sum_{i=1}^{r} x_{ik}e_{ik}$, the corresponding processor reliability can be computed by

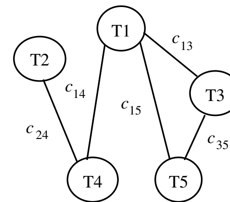

Fig. 2. An example of TIG.

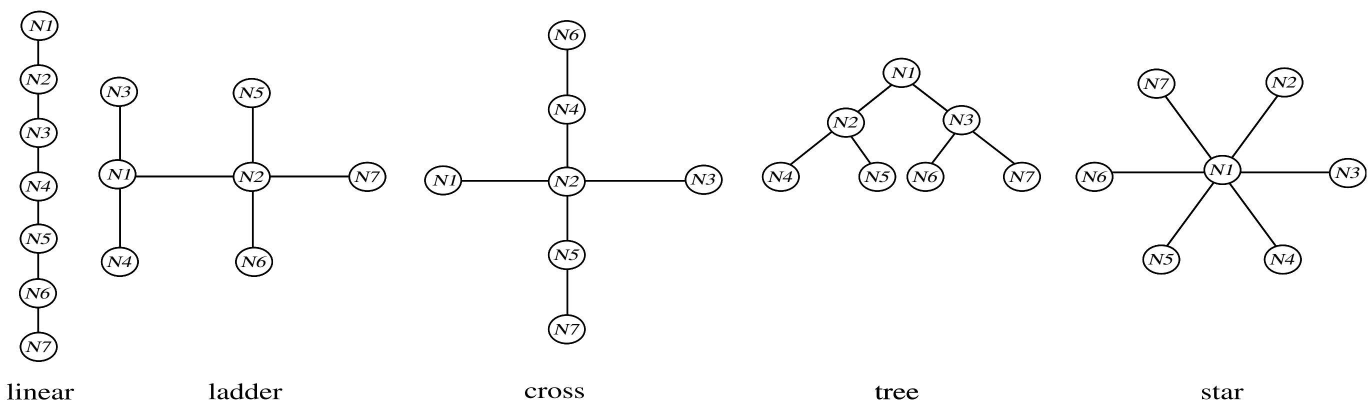

Fig. 1. An example of PIG.

$$R_k(\boldsymbol{X}) = \mathrm{e}^{-\lambda_k \sum_{i=1}^{r} x_{ik} e_{ik}}. \tag{1}$$

Similarly, the reliability for the communication link l_{kb} during a time interval t is $R_{kb}(\boldsymbol{X}) = \mathrm{e}^{-\mu_{kb} t}$. Since the total elapsed time for transmitting the intermodule communication cost via l_{kb} is $\sum_{i=1}^{r}\sum_{i\neq j} x_{ik}x_{jb}(c_{ij}/w_{kb})$, the corresponding communication link reliability is given by

$$R_{kb}(\boldsymbol{X}) = \mathrm{e}^{-\mu_{kb}\sum_{i=1}^{r}\sum_{i\neq j} x_{ik}x_{jb}(c_{ij}/w_{kb})}. \tag{2}$$

As the system reliability requires that all involved components are operational during the elapsed time for the execution, the DSR with the task allocation $\boldsymbol{X}$ is computed as follows:

$$R(\boldsymbol{X}) = \prod_{k=1}^{n} R_k(\boldsymbol{X}) \prod_{k=1}^{n-1}\prod_{b>k} R_{kb}(\boldsymbol{X}). \tag{3}$$

Maximizing the DSR is equivalent to minimizing the following cost,

$$\mathrm{COST}(\boldsymbol{X}) = \sum_{k=1}^{n}\sum_{i=1}^{r} \lambda_k x_{ik} e_{ik} + \sum_{k=1}^{n-1}\sum_{b>k}\sum_{i=1}^{r}\sum_{i\neq j} \mu_{kb} x_{ik} x_{jb} (c_{ij}/w_{kb}). \tag{4}$$

With the system resource constraints described in our assumptions, the addressed task allocation problem is formulated by the following 0–1 quadratic programming problem:

$$\text{Min} \quad \mathrm{COST}(\boldsymbol{X}), \tag{5}$$

$$\text{subject to} \quad \sum_{k=1}^{n} x_{ik} = 1 \quad \forall i = 1,2,\ldots,r, \tag{6}$$

$$\sum_{i=1}^{r} m_i x_{ik} \leqslant M_k \quad \forall k = 1,2,\ldots,n, \tag{7}$$

$$\sum_{i=1}^{r} s_i x_{ik} \leqslant S_k \quad \forall k = 1,2,\ldots,n, \tag{8}$$

$$x_{ik} \in \{0,1\} \quad \forall i,k. \tag{9}$$

Constraint (6) states that each module should be assigned to exactly one processor. Constraints (7) and (8) ensure that the memory and computation resource capacity of each processor is no less than the total amount of resource requirements of all of its assigned modules. Constraint (9) guarantees that x_{ik} are binary variables.

The above formulation is a 0–1 programming problem with a quadratic objective function and it is known to be NP-hard (Lin and Chen, 1997). Solving the problem is computationally prohibitive due to enormous computations. Although exact algorithms such as branch and bound and A* algorithms have been proposed, the worst-case complexity is still exponential and they are limited to small task allocation problems. An alternative is to find approximate solutions efficiently using metaheuristics. In this paper, we propose a hybrid PSO for this purpose.

3. Proposed algorithm

Particle swarm optimization (PSO) has been employed to cope with the project scheduling (Zhang et al., 2006; Zhang et al., 2005) and flow-shop scheduling (Tasgetiren et al., 2004; Xia and Wu, 2005) problems, but not previously to the task allocation problem considered in this paper. The project scheduling and flow-shop scheduling problems focus in finding a permutation of project activities or jobs to be processed by a set of machines and the permutation may be subject to precedence constraints. The priority-based solution representation is usually adopted to enumerate the permutation. While the task allocation problem aims at finding an assignment of modules to a set of processors subject to the resource constraints. A new solution representation needs to be devised and the infeasible solutions generated should be carefully handled.

3.1. Particle swarm optimization

The particle swarm optimization (PSO) algorithm was proposed by Kennedy and Eberhart (1995). It is inspired by the behavior of bird flocking and fish schooling: a large number of birds/fishes flock synchronously, change direction suddenly, scatter and regroup iteratively, and finally perch on a target. The PSO algorithm mimics such interesting behavior and serves as a function optimizer: a swarm of particles/solutions are randomly initialized and each individual improves by referring to the experience of its own and that of the entire swarm. The swarm intelligence is enriched along with the evolution of the particles and thus the near-optimal solution can be found. The convergence and parameterization aspects of the PSO are discussed in Clerc and Kennedy (2002) and Trelea (2003). The attractiveness of the PSO includes the following features: natural metaphor, stochastic move, adaptivity, and positive feedback.

The principal components of the PSO are outlined as follows.

- *Particle representation*: The particle is represented as a vector of real random variables that characterize the problem. Let the problem solution be described by r variables, we denote the ith particle by $P_i = (p_{i1}, p_{i2}, \ldots, p_{ir})^{\mathrm{T}} \in R^r$. Thus, each particle is a candidate solution to the optimization problem and it changes values iteratively according to different forms of experiences.
- *Swarm*: The PSO is a population-based searching paradigm that explores the solution space by flying a number of particles, called a swarm. The initial swarm is generated at random and the size of a swarm is usually kept constant throughout iterations.
- *Experience*: The major feature of PSO is that it is an elegant way of managing experiences. The swarm intelligence is enriched by tallying the best experiences observed by individuals and the entire swarm. The

personal best experience is employed to update the global experience, and the global experience is used in guiding the flying of each particle. This is a positive feedback process that increases the success rate in foraging for the social system, or in the optimization language, it increases the probability for targeting the optimal solution.
- *Stopping criterion*: To obtain quality solution, we need to run the PSO algorithm until it converges and the final best solution found so far is output. Hence, a stopping criterion that can identify the convergence behavior of the algorithm is needed. The widely-used stopping criteria for evolutionary algorithms are as follows: (1) the algorithm is terminated after a fixed and sufficiently large number of iterations, (2) if the number of experienced iterations between two consecutive global best solution improvements exceeds a fixed number, or (3) if the last k improvements over the global best solution are negligible. Practitioners can use either or combination of these criteria in accordance with their purposes and applications.

3.2. PSO for the task allocation problem

In the following, we propose a PSO algorithm for tackling the addressed task allocation problem described in Section 2. Each component of the PSO is customized to the problem. A hybrid approach that combines the PSO and a heuristic is finally developed for expediting the convergence speed.

3.2.1. Particle representation and initial swarm generation

Each particle should represent a solution for the task allocation problem, i.e., we can obtain an instance of task allocation from the particle representation directly or indirectly. An *intuitive representation* of a particle could include all decision variables, x_{ik}, $\forall i = 1,2,\ldots,r$, $\forall k = 1,2,\ldots,n$, however, this representation is not efficient since most of the decision variables are equal to zero and Constraint (6), which states that each module should be assigned to exactly one processor, will make the modification of the particle difficult.

Instead, we represent each particle with a vector of r elements, and each element is an integer value between 1 and n. Fig. 3 shows an illustrative example for the ith particle P_i, which describes a task allocation that assigns five modules to three processors. For example, $p_{i4} = 1$ means that the fourth module is assigned to the first processor. Formally, $p_{i,j} = k$ implies that $x_{jk} = 1$ and $x_{jl} = 0$, $\forall l \neq k$. Our particle representation is n times more condensed than the intuitive representation of the particle, and Constraint (6) is always satisfied during the modification of the particle values.

	p_{i1}	p_{i2}	p_{i3}	p_{i4}	p_{i5}
P_i :	2	3	1	1	3

Fig. 3. An example for the ith particle.

The PSO approach generates randomly an initial swarm of N particles, where N is the swarm size. These particle vectors will be iteratively modified according to collective experiences in order to improve their solution quality.

3.2.2. Fitness evaluation

The particles compete for the best solution during the evolution according to a measure of solution quality, called *fitness*, such that the swarm evolution is navigated towards the optimal solution by the best particles. The original objective function COST($\boldsymbol{X}$) of the problem measures the merit of a given solution $\boldsymbol{X}$ according to the assumption that this solution meets all the Constraints 6–9. However, the value of COST($\boldsymbol{X}$) is discredited if the solution violates at least one of the constraints, i.e., if the solution is infeasible. For the constraints directly related to the solution representation, it is possible to design specific operations for generating solutions that are always valid. For example, Partially Matched Crossover (PMX) (Goldberg and Lingle, 1985) has been broadly used in GA to ensure that the resulting solution represents a permutation because the validity can be directly checked from the solution. However, for the TAP considered in this paper, Constraints (7) and (8) are indirectly imposed on the solution, we need to first compute the amount of consumed memory and computation resource from the whole solution (all decision variables), and then we are able to check the solution validity if the amount of consumed resource exceeds the capacity. We have no knowledge about the validity until we have completely generated the solution. Hence, it is very difficult to devise an operation for always generating valid solutions in this case. Alternatively, researchers have resorted to designing penalty functions for decreasing the fitness of invalid solutions, if yielded.

We have devised penalty functions to estimate the infeasibility of a particle solution. Since our particle representation indicates the index of the allocated processor of each module in one element (see Fig. 3), the satisfaction of Constraint (6) ensuring that each module is assigned to exactly one processor is guaranteed. Furthermore, from the formal definition of our particle representation, $p_{i,j} = k$ implies that $x_{jk} = 1$ and $x_{jl} = 0$, $\forall l \neq k$, Constraint (9) stating that the decision variable x_{ij} takes value of either 0 or 1 is also satisfied. Hence, the penalty functions estimating the infeasibility of a particle solution are only related to Constraints (7) and (8), and they are defined as follows.

- Penalty function α for violating Constraint (7)

$$\alpha(\boldsymbol{X}) = \sum_{k=1}^{n} \max\left(0, \sum_{i=1}^{r} m_i x_{ik} - M_k\right). \tag{10}$$

The term $\max\left(0, \sum_{i=1}^{r} m_i x_{ik} - M_k\right)$ calculates the insufficiency amount of memory resource on processor k if the

memory requirement incurred by all the running modules allocated to processor k exceeds the memory capacity of the processor. Otherwise, it returns zero. The penalty function $\alpha(X)$ then sums the total insufficiency amounts of memory over all available processors. Thus, the penalty function gives proportional amount of penalties on the infeasible solution whose requested memory resource exceeds the system memory capacity. The more the memory insufficiency is, the greater the penalty is.

- Penalty function β for violating Constraint (8)

$$\beta(\boldsymbol{X}) = \sum_{k=1}^{n} \max\left(0, \sum_{i=1}^{r} s_i x_{ik} - S_k\right). \tag{11}$$

Similarly, this function calculates the total insufficiency amount of computation resource over all available processors.

The penalty functions are then combined with the original objective function in a weighed manner.

$$J(\boldsymbol{X}) = \mathrm{COST}(\boldsymbol{X}) + \delta_1\alpha(\boldsymbol{X}) + \delta_2\beta(\boldsymbol{X}), \tag{12}$$

where δ_1 and δ_2 are the weights controlling the relative significance of respective penalty functions. Those weights are determined automatically in our method according to two criteria. *First*, they should scale the possible values of $\alpha(\boldsymbol{X})$ and $\beta(\boldsymbol{X})$ to comparable ranges as that of $\mathrm{COST}(\boldsymbol{X})$ such that the evolution trend will be influenced by the penalty incurred and move towards valid solutions and away from invalid ones. In our preliminary experiments, the value of $\mathrm{COST}(\boldsymbol{X})$ is always below 1.0 during evolution. The ranges of $\alpha(\boldsymbol{X})$ and $\beta(\boldsymbol{X})$ can be estimated using a sufficient number of solutions generated at random (we use 100 random solutions), because once the evolution started, the values of $\alpha(\boldsymbol{X})$ and $\beta(\boldsymbol{X})$ decrease, and they become zero when the yielded solution is valid. *Second*, the weights can be further tuned if the decision-maker has different preference of respective constraints. For example, if the memory resource is considered to be cheaper than the computation resource and can be augmented if necessary, we may decrease the value of δ_1 and increase the value of δ_2. In this paper, we assume that the two constraints have equal importance.

From the above discussions, we know that the larger the value of $J(\boldsymbol{X})$, the worse the quality of solution $\boldsymbol{X}$ is. Since the fitness function measures the merit of the particle, we define the fitness function as follows:

$$\mathrm{Fitness}(\boldsymbol{X}) = J(\boldsymbol{X})^{-1}. \tag{13}$$

3.2.3. Particle vector modification according to experiences

As described in Section 3.1, the swarm intelligence is enriched by tallying the best experiences observed by individual particles and the entire swarm during the evolution. In particular, each particle remembers the best vector with the highest fitness it visited so far, referred to as *pbest*, and the best vector visited by its neighbors. There are two versions for keeping the neighbors' best vector. In the local version, each particle keeps track of the best vector *lbest* attained by the particles within its local neighborhood according to Euclidean distance in the solution space. For the global version, the best vector *gbest* is determined by any particles in the entire swarm.

The particle vector modification formula has several variations (Kennedy and Eberhart, 1995; Clerc and Kennedy, 2002; Trelea, 2003; Kennedy et al., 2001). Among them, Clerc and Kennedy (2002) has proved that the use of a constriction factor is needed to insure the convergence of the algorithm. It proceeds as follows. At each evolutionary iteration, the ith particle modifies its velocity v_{ij} and position p_{ij} through each dimension j by referring to, with random multipliers, the personal best experience ($pbest_{ij}$) and the swarm's best experience ($gbest_j$) using Eqs. (14) and (15) as follows:

$$v_{ij} \leftarrow K[v_{ij} + \xi_1 rand_1(pbest_{ij} - p_{ij}) + \xi_2 rand_2(gbest_j - p_{ij})], \tag{14}$$

$$p_{ij} \leftarrow p_{ij} + v_{ij}, \tag{15}$$

where ξ_1 and ξ_2 are the cognitive coefficients and $rand_1$ and $rand_2$ are random real numbers drawn from $U(0,1)$. K is the constriction coefficient and is given by

$$K = \frac{2}{|2 - (\xi_1 + \xi_2) - \sqrt{(\xi_1 + \xi_2)^2 - 4(\xi_1 + \xi_2)}|} \quad \text{s.t. } \xi_1 + \xi_2 > 4. \tag{16}$$

Typically, setting $\xi_1 = \xi_2 = 2.05$ is applicable to many domains and K is thus 0.729.

It is observed from Eqs. (14) and (15) that the particle will fly through the solution space navigated by $pbest_i$ and *gbest* while still exploring new areas by the stochastic mechanism to escape from the barrier of local optimality. Our algorithm is terminated with a given maximum number of iterations and the final solution delivered by *gbest* is reported.

3.2.4. Hybrid PSO

Modern metaheuristic algorithms perform both the *exploitation* and *exploration* search components in program iterations. Exploration involves searching for new regions in the solution space, and once it finds a good region exploitation involves searching for local optima. For example, genetic algorithms (Goldberg, 1989) employ selection and crossover operators as exploitation search and mutation operator as exploration search. The ant colony optimization (ACO) (Dorigo and Gambardella, 1997) also devises visibility (exploration) and pheromone (exploitation) components in one search framework. Analogously, the original particle swarm optimization (PSO) (Kennedy et al., 2001) also interweaves these two parts. The exploration search of new possibilities is fulfilled by the cognitive coefficients (ξ_1 and ξ_2) and the randomness multipliers ($rand_1$ and $rand_2$), while the exploitation search is performed by referring to best experiences (*pbest* and *gbest*) found so

far (see Eq. (14)). Higher values of coefficients and multipliers will increase the oscillation frequency of particles. However, as the multipliers are generated at random, the weighting between exploration and exploitation changes at every iteration. There exists another PSO variation (Coello Coello et al., 2004) which embeds the mutation operator into the iterations to emphasize the explorative behavior for solving the multi-objective optimization. However, in our preliminary experiments, the addition of the mutation operator into our PSO algorithm does not significantly benefit the produced result. This is probably because our task allocation problem is formulated as single-objective optimization.

It has been shown in many applications that a hybrid strategy that combines a metaheuristic algorithm with a local heuristic can obtain a better result and converge more quickly (Michalewicz and Fogel, 2002). Therefore, we devise a parameter-wise hill-climbing heuristic for embedding into the proposed PSO algorithm as follows. Given a particle vector, its solution quality can be improved locally by scanning the particle elements for updating. As in our particle representation each particle consists of r elements and the value of each element ranges from 1 to n, we sequentially scan each element and look for possible replacement with other values if the particle fitness is improved. When an element is examined, the values of the remaining $r-1$ elements remain unchanged. The heuristic is terminated if all the elements of the particle have been examined for updating. The computation for the fitness value due to the element updating can be minimized since a value change in one element affects the allocation of exactly one module, we can save the fitness computation by only recalculating the system reliability and constraint conditions related to the reallocated module.

The proposed hybrid PSO algorithm (which shall be referred to as HPSO) is summarized in Fig. 4. At the initialization step, the algorithm prepares an initial swarm of N particles and a set of initial velocities generated at random. At the iteration step, all of the particles modify repeatedly their vectors until a maximal number of iterations have passed. During each iteration, the personal best and swarm's best vectors are identified. The particle then adjusts its vector using Eqs. (14) and (15) according to the constriction factor model. To expedite the convergence speed, all of the particles are further improved using the parameter-wise hill-climbing heuristic before getting into the next iteration.

4. Experimental results and discussion

To evaluate the efficiency and effectiveness of the proposed algorithm, intensive experiments have been conducted. A large simulation dataset is created which features different problem size, task interaction density, and network topology. The proposed algorithm is compared with a genetic algorithm (GA), a heuristic approach, and an exact method. All of the experiments are performed on a 2.4 GHz PC with 256 MB RAM. Analysis of convergence and worst-case issues is also presented.

4.1. Experimental scenarios

To assess the comparative performances of the proposed HPSO algorithm with other algorithms, a large simulation dataset is created. The dataset fits very much the real world DCS by taking into account different numbers of processors (n) and modules (r), and different characteristics of PIGs and TIGs. First, we set the value of (n,r) to be equal to $(6,8)$, $(6,12)$, $(7,9)$, $(7,13)$, $(8,10)$, $(8,16)$, $(9,11)$, and $(9,17)$, respectively, in order to testify the algorithm with different problem sizes. Secondly, for each pair of (n,r), we consider three different TIGs with various task interaction density d equal to 0.3, 0.5, and 0.8. Thirdly, five types of typical PIG topologies, namely linear, ladder, cross, tree, and star, as shown in Fig. 1 are simulated. Therefore, there are totally $8\times3\times5=120$ DCS instances in our testing

1. Initialize.
   - 1.1 Generate an initial swarm of N particles at random.
   - 1.2 Generate initial velocities v_{ij}, $1 \le i \le N$ and $1 \le j \le r$, at random.
2. Repeat until a given maximal number of iterations is achieved.
   - 2.1 Evaluate the fitness of each particle using Eq. (13).
   - 2.2 Determine the best vector *pbest* visited so far by each particle.
   - 2.3 Determine the best vector *gbest* visited so far by the whole swarm.
   - 2.4 Update velocities v_{ij} using Eq. (14) according to the constriction factor model.
   - 2.5 Update particle vectors using Eq. (15).
   - 2.6 Improve the solution quality of each particle using the parameter-wise hill-climbing heuristic.

Fig. 4. Summary of the HPSO algorithm.

Table 2
System parameters and the corresponding testing ranges

System parameters	Testing ranges
Failure rate of processor	[0.00005, 0.00010]
Failure rate of communication link	[0.00015, 0.00030]
Accumulative execution time (AET)	[15, 25]
Intermodule communication (IMC) cost	[15, 25]
Memory resource requirement	[1, 60]
Computation resource requirement	[1, 60]
Memory resource capacity	[100, 200]
Computation resource capacity	[100, 200]

dataset that assess deliberately the performance of competing algorithms against different problem size, task interaction density, and network topology. The values of other system parameters are generated randomly with the ranges listed in Table 2.

4.2. Comparative performances

To evaluate the efficiency and effectiveness of the proposed HPSO algorithm, we have implemented an exact method, a heuristic approach, and an approximation method. The exact method solves the 0–1 quadratic programming problem as given in Eqs. (5)–(9) using the commercial Lingo software package (Lingo System). When Lingo fails to solve the problem instance within 24 h, it is terminated with an infeasible solution or an unknown status and we discard such cases for further comparison. The heuristic approach developed from Kartik and Murthy (1997) reorders the modules in a way that the last r modules do not communicate with one another, i.e., they are communication-independent. Then, a state-space tree is established to enumerate the allocations of modules. At each level of the tree, the allocation of the module from the ordered list is enumerated and the node with the least cost estimated is chosen to expand next. We implemented this heuristic approach because it has been shown to be superior to several other heuristics (Kartik and Murthy, 1997).

The approximation method is developed from a GA which uses the same coding scheme (see Fig. 3) and fitness function (see Eq. (13)) as in the HPSO. The GA evolves with a population of 80 chromosomes and the crossover and mutation rates are set as 0.7 and 0.1, respectively. These GA parameter values are determined experimentally with the following ranges. The population size varies from 1 to 100 with increment of 20. The crossover and mutation rates are both tested within $(0, 1)$ with increment of 0.1. The number of fitness evaluations until termination is set to be 80,000 according to the observation that GA can converge and reach stagnation before this number of fitness evaluations in all of the experiments.

In all our experiments, we terminate HPSO and GA when they have experienced 80,000 times of fitness evaluations because we want to assess the comparative performance of the two algorithms, we need to fix one performance measure (the consumed computational time) and compare the other (the derived system reliability). Furthermore, both HPSO and GA are stochastic-based algorithms and each independent run of the same algorithm on a particular problem instance may yield a different result, we thus report the average reliability and computational time over 10 independent runs for the two algorithms.

From the intensive experiments we find that the network topology does not have measurable impact on the results, we thus only tabulate the detailed computational results for one topology and then provide the overall statistics. Table 3 shows the distributed system reliability (DSR) and the computational time (t) obtained using different algorithms tested on the tree topology. The approximation rate (Φ) defined as the approximation reliability divided by the optimal reliability is also provided for a quick glance of how close the approximation methods converge to the exact solutions. From the results, we have the following observations:

Efficiency: The CPU times used by Lingo for deriving the exact optimal reliability grow rapidly with the problem complexity depending on the parameters (n, r, d). This limits the applicability of Lingo to problems of small size. For all the 120 problem instances of various network topologies in our dataset, Lingo fails to solve 47 problems when the CPU times used have exceeded 24 h, indicating the need for heuristic or metaheuristic algorithms which can derive approximation solutions to large problems within reasonable times. As can be seen from Table 3, the CPU times consumed by HPSO and GA for obtaining near-optimal reliability ranges from 1.4 to 22.8 s for all tested problem sizes, the computational saving compared with that of Lingo is remarkable. Moreover, the HPSO always consumes less CPU time than GA, although both algorithms are set to run for the same number of fitness evaluations. This is because in the embedded heuristic of the HPSO, the computation for fitness value can be expedited by calculating only the system reliability and constraint conditions related to the reallocated module. Furthermore, the heuristic developed from Kartik and Murthy (1997) is the fastest method among all competing algorithms. The CPU times required ranges from 0.02 to 0.06 s. However, the DSR derived is not satisfactory as the problem size increases.

Effectiveness: Compared with Lingo, the HPSO and GA can maximize the reliability as much as possible through evolutionary computation. The approximation rate (Φ) of the reliability obtained by the HPSO to the exact optimal reliability ranges from 99.52% to 100.0%, while it is between 96.10% and 100.0% for the GA, indicating that the HPSO is more effective than the GA on the test-cases studied. The experimental results for other network topologies also support this claim. Moreover, the HPSO algorithm obtains the exact optimal reliability for 22 out of the 120 tested problem instances, which is 18.3% from the dataset, while the GA finds exact optimal reliability

Table 3
The reliability, approximation rate and computational time obtained using different algorithms tested on the tree network topology

n	*r*	*d*	HPSO			GA			Kartik and Murthy (1997)			LINGO	
			DSR	Φ (%)	t	DSR	Φ (%)	*t*	DSR	Φ (%)	*t*	DSR	*t*
6	8	0.3	0.982	99.97	1.6	0.982	99.95	11.8	0.942	95.82	0.04	0.983	29
		0.5	0.981	100.00	1.6	0.980	99.91	11.6	0.911	92.88	0.03	0.981	65
		0.8	0.970	99.97	1.4	0.970	99.97	12.7	0.855	88.12	0.06	0.971	56
	12	0.3	0.964	99.54	3.2	0.953	98.38	14.4	0.864	89.23	0.05	0.969	494
		0.5	0.926	99.59	3.4	0.917	98.70	15.2	0.759	81.69	0.04	0.929	7186
		0.8	0.939	100.00	3.6	0.933	99.36	14.8	0.676	71.98	0.04	0.939	779
7	9	0.3	0.985	100.00	2.2	0.985	100.00	12	0.922	93.56	0.03	0.985	125
		0.5	0.985	99.95	2.4	0.985	99.95	12.2	0.918	93.17	0.05	0.985	43
		0.8	0.948	100.00	2.6	0.948	99.97	12.6	0.820	86.44	0.04	0.948	575
	13	0.3	0.971	NA	4.6	0.963	NA	15.2	0.933	NA	0.05	NA	NA
		0.5	0.917	NA	4.8	0.892	NA	15.6	0.784	NA	0.06	NA	NA
		0.8	0.863	NA	5.2	0.806	NA	16.4	0.578	NA	0.02	NA	NA
8	10	0.3	0.986	99.98	3.4	0.985	99.86	13	0.900	91.25	0.03	0.986	812
		0.5	0.972	99.80	3.4	0.970	99.54	13.2	0.869	89.23	0.05	0.974	1830
		0.8	0.903	99.52	3.6	0.889	97.96	14.4	0.699	77.08	0.06	0.907	29000
	16	0.3	0.914	NA	8.8	0.880	NA	18.6	0.767	NA	0.05	NA	NA
		0.5	0.904	NA	9.6	0.873	NA	19.6	0.712	NA	0.05	NA	NA
		0.8	0.825	NA	10.8	0.708	NA	20.8	0.308	NA	0.02	NA	NA
9	11	0.3	0.978	99.94	5.2	0.976	99.69	14.8	0.916	93.61	0.05	0.979	1850
		0.5	0.967	99.58	5.4	0.943	97.18	15.2	0.849	87.40	0.02	0.971	6391
		0.8	0.915	99.77	5.8	0.881	96.10	16	0.543	59.22	0.04	0.917	30457
	17	0.3	0.895	NA	12.8	0.846	NA	20.8	0.679	NA	0.05	NA	NA
		0.5	0.822	NA	13.2	0.749	NA	21.2	0.547	NA	0.05	NA	NA
		0.8	0.692	NA	14.4	0.576	NA	22.8	0.337	NA	0.05	NA	NA

NA: not available.

for 10 out of 120 problem instances, which is less than half of the hit ratio achieved by the HPSO. For all problems in the dataset, the reliability obtained by the HPSO is always higher or equal to that derived by the GA. As for the heuristic developed from Kartik and Murthy (1997), although it is the fastest method among all the competing algorithms, the DSR derived is always lower than that obtained by the HPSO and GA, and the difference becomes larger with increasing problem size. The approximation rate (Φ) ranges from 59.22% to 95.82%, which is also significantly lower than that obtained by the HPSO and GA.

Table 4 gives some overall statistics that characterize the metaheuristics. The means and standard deviations on reliability and computational time are reported. As can be seen, when the task interaction density d increases, the intermodule communication becomes more intensive and the reliability derived by the HPSO and GA could be lower due to involvement of more communication links. Moreover, the CPU time required increases slightly with large d because of the extra computations for the reliability related to the communication links involved. On the other hand, the HPSO is more stable on the reliability produced for the five tested topologies. From the overall statistics, we see that the standard deviation for all independent runs of the HPSO is 0.064, while it is 0.092 for the GA, and the difference on the overall mean reliability is more than 2.2%.

In summary, the proposed HPSO algorithm is *effective* because it can derive exact optimal solutions for 18.3% of

Table 4
The means and standard deviations on reliability and computational time

Statistics	HPSO				GA			
	Reliability		CPU time		Reliability		CPU time	
	Mean	Std. dev.	Mean	Std. dev.	Mean	Std. dev.	Mean	Std. dev.
$d = 0.3$	0.95964	0.028	7.0	4.321	0.94682	0.043	23.0	5.353
$d = 0.5$	0.93085	0.061	7.2	4.511	0.91044	0.086	23.9	5.643
$d = 0.8$	0.88886	0.079	7.8	5.048	0.85358	0.116	24.4	5.761
Linear	0.92652	0.073	7.2	3.659	0.90213	0.099	27.9	5.907
Ladder	0.92525	0.070	9.1	6.117	0.89976	0.103	23.4	6.747
Cross	0.93163	0.059	9.3	5.335	0.90994	0.084	25.9	6.194
Tree	0.92525	0.070	5.5	3.907	0.89961	0.102	15.6	3.314
Star	0.92361	0.070	5.6	3.890	0.90662	0.089	25.9	6.664
Overall	0.92645	0.064	7.3	4.444	0.90361	0.092	23.7	5.373

the dataset and the overall approximation rate is very high and stable against different DCS scenarios. The HPSO is also *efficient* because it reports the results in just a few seconds, while Lingo fails to solve almost 40% of the dataset within reasonable time.

4.3. Convergence analysis

The convergence analysis of the HPSO algorithm should rely on the entire swarm not just on a lucky particle. The particles are inter-communicated by sharing their best experiences, which makes the main feature of PSO different from the random walk. Hence, we examine whether all the personal best experiences evolve to the same target and finally obtain a high-quality solution.

We use the information *entropy* for measuring the similarity among all *pbest* experiences as the evolution proceeds. Let $prob_j(s)$ be the probability that a *pbest* assigns the jth module to the sth processor, *viz*.

$$prob_j(s) = \Pr\{pbest_{ij} = s | i = 1, 2, \ldots, N\},\quad \forall s = 1, 2, \ldots, n. \tag{17}$$

The information entropy over the probability distributions $prob_j(s)$, $s = 1, 2, \ldots, n$, is given by

$$\Psi_j = -\sum_{s=1}^{n} prob_j(s) \log_2(prob_j(s)). \tag{18}$$

Ψ_j describes the degree of consensus over all *pbest* for the decision on the allocation of the jth module. To provide an overall measure, we compute the average particle entropy as

$$\bar{\Psi} = \sum_{s=1}^{r} \Psi_j / r. \tag{19}$$

The smaller the value of $\bar{\Psi}$, the higher the degree of consensus among all *pbest* is.

Fig. 5 shows the variations of the average particle entropy ($\bar{\Psi}$) during the HPSO evolution. We observe that during the first 20,000 fitness evaluations, the entropy value decreases rapidly. This is because at the early evolution stage, the particles widely explore the possible allocation of each module and share their best experiences. The particles are quickly resorting to the good-quality regions in the solution space, thus resulting in a fast decreasing rate for $\bar{\Psi}$ at this stage. During the period between 20,000 and 50,000 fitness evaluations, the particles exploit the neighboring solutions around the good-quality regions, only few elements of the particle will be updated since most elements of *pbest* are identical to that of *gbest*. Hence, the decreasing rate for $\bar{\Psi}$ becomes gentle at this stage. After 50,000 fitness evaluations, the particles are more similar to one another, the decrement of $\bar{\Psi}$ almost stagnates. We conclude from Fig. 5 that all *pbest* are resorting to the same target as the algorithm proceeds due to the collective awareness from the entire swarm instead of the movement of a lucky particle.

4.4. Worst-case analysis

Since the HPSO is a stochastic algorithm, each separate run of the program could yield a different result. We still need to argue that the high-quality solution delivered by the algorithm is not due to a lucky run, but a consequence of the evolution. The worst-case analysis, which is set up as the worst optimal solution we could get after a specific number of repetitive runs of the HPSO, can provide a good argument. Fig. 6 shows the worst-case analysis where the HPSO is executed for 1000 times on a problem instance from the tree topology network where $(n, r, d) = (8, 10, 0.8)$. We observe that the HPSO can derive a task allocation with system reliability higher than 0.90 (the optimal reliability is 0.907375 as shown in Table 3) after 65 repetitive runs of the proposed algorithm in the worst case. However, in the general case, the HPSO is likely to derive such a good solution with much fewer runs, because the probability that each separate run of the HPSO will obtain a solution with reliability higher than 0.90 is 93.5% (estimated by 1 − 65/1000). Therefore, the high-quality solution delivered by the algorithm is not due to a lucky run, but a consequence of the evolution.

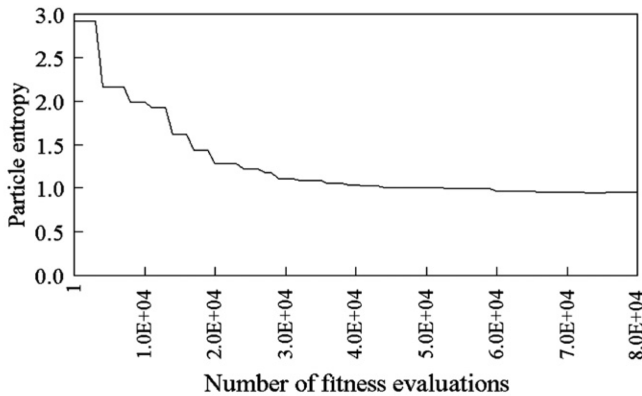

Fig. 5. The particle entropy ($\bar{\Psi}$) over all *pbest* versus the number of fitness evaluations.

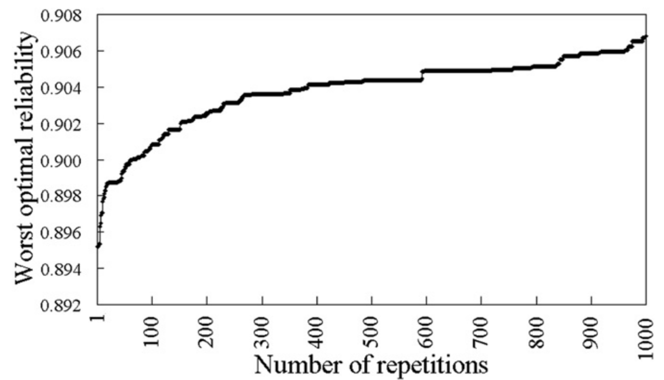

Fig. 6. The worst-case analysis versus the number of repetitive runs of HPSO.

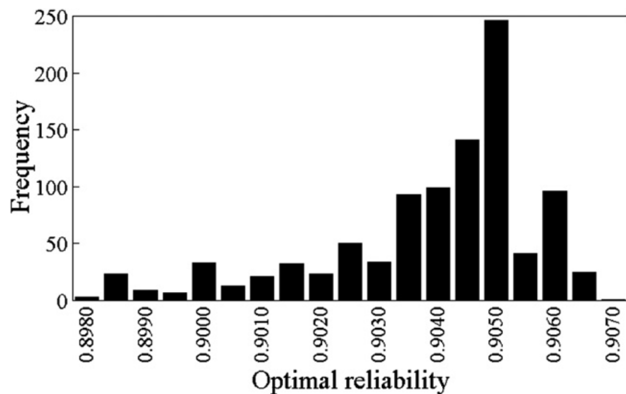

Fig. 7. The reliability values obtained from repetitive runs of HPSO follow a quasi-normal distribution.

We also observe that the reliability values obtained by those repetitive runs follow a quasi-normal distribution with mean and standard deviation equal to 0.9038 and 0.0019 respectively (see Fig. 7). Those reliability values are very close to the exact optimal reliability (0.907375) and are significantly higher than the mean reliability (0.888874) obtained by the GA. Thus, we can justify that the proposed HPSO algorithm has good control over the evolution process and does not suffer premature convergence.

5. Conclusions

In this paper, we have proposed a hybrid particle swarm optimization (HPSO) algorithm which maximizes the distributed system reliability (DSR) of executing successfully a task consisting of several modules. The HPSO initializes a swarm of particles each of which corresponds to a candidate solution to the underlying problem. These particles employ their personal best experience to update the swarm experience, and the swarm experience is used in guiding the flying of each particle. This is a positive feedback process such that the fitness of particles is improved. Penalty functions tailored to the system constraints are devised in order to deal with infeasible solutions. The HPSO embeds a local search heuristic into the evolutionary iterations for expediting the convergence. The experimental results manifest that the HPSO reports quality solutions on a large set of simulated instances involving different problem sizes, task interaction densities, and network topologies. The proposed method is also more effective and efficient than a genetic algorithm on the tested dataset. The convergence and the worst-case analyses of the HPSO have been theoretically and empirically conducted.

References

Clerc, M., Kennedy, J., 2002. The particle swarm explosion, stability, and convergence in a multidimensional complex space. IEEE Transaction on Evolutionary Computation 6, 58–73.

Coello Coello, C.A., Pulido, G.T., Lechuga, M.S., 2004. Handling multiple objectives with particle swarm optimization. IEEE Transaction on Evolutionary Computation 8, 256–279.

Dorigo, M., Gambardella, L., 1997. Ant colony system: a cooperative learning approach to the traveling salesman problem. IEEE Transaction on Evolutionary Computation 1, 53–66.

Ernst, A., Hiang, H., Krishnamoorthy, M., 2001. Mathematical programming approaches for solving task allocation problems. In: Proceedings of the 16th National Conference Of Australian Society of Operations Research.

Glover, F., 1989. Tabu search – Part I. ORSA Journal of Computing 1, 190–206.

Goldberg, D.E., 1989. Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley, Reading, MA.

Goldberg, D.E., Lingle, R., 1985. Alleles, loci, and the traveling salesman problem. In: Proceedings of an International Conference on Genetic Algorithms and Their Applications, pp. 154–159.

Hsieh, C.C., 2003. Optimal task allocation and hardware redundancy policies in distributed computing systems. European Journal of Operational Research 147, 430–447.

Hsieh, C.C., Hsieh, Y.C., 2003. Reliability and cost optimization in distributed computing systems. Computers and Operations Research 30, 1103–1109.

Kartik, S., Murthy, S.R., 1995. Improved task-allocation algorithms to maximize reliability of redundant distributed computing systems. IEEE Transactions on Reliability 44, 575–586.

Kartik, S., Murthy, S.R., 1997. Task allocation algorithms for maximizing reliability of distributed computing systems. IEEE Transactions on Computers 46, 719–724.

Kennedy, J., Eberhart, R.C., 1995. Particle swarm optimization. Proceedings of the IEEE International Conference on Neural Networks IV, 1942–1948.

Kennedy, J., Eberhart, R.C., Shi, Y., 2001. Swarm Intelligence. Morgan Kaufmann Publishers, London.

Kumar, V.K.P., Raghavendra, C.S., Hariri, S., 1986. Distributed program reliability analysis. IEEE Transactions on Software Engineering 12, 42–50.

Lee, C.H., Shin, K.G., 1997. Optimal task assignment in homogeneous networks. IEEE Transactions on Parallel and Distributed Systems 8, 119–129.

Lin, M.S., Chen, D.J., 1997. The computational complexity of the reliability problem on distributed systems. Information Processing Letters 64, 143–147.

Lingo System. Available from: <http://www.lindo.com/>.

Michalewicz, Z., Fogel, D.B., 2002. How to Solve it: Modern Heuristics. Springer-Verlag.

Ajith, T.P., Murthy, C.S.R., 1999. Optimal task allocation in distributed systems by graph matching and state space search. Journal of Systems and Software 46, 59–75.

Shatz, S.M., Wang, J.P., Goto, M., 1992. Task allocation for maximizing reliability of distributed computer systems. IEEE Transactions on Computers 41, 1156–1168.

Shigenori, N., Takamu, G., Toshiku, Y., Yoshikazu, F., 2003. A hybrid particle swarm optimization for distribution state estimation. IEEE Transaction on Power Systems 18, 60–68.

Sinnen, O., Sousa, L.A., Sandnes, R.E., 2006. Toward a realistic task scheduling model. IEEE Transactions on Parallel and Distributed Systems 17, 263–275.

Srinivasan, S., Jha, N.K., 1999. Safety and reliability driven task allocation in distributed systems. IEEE Transactions on Parallel and Distributed Systems 10, 238–251.

Tasgetiren, M.F., Sevkli, M., Liang, Y.C., 2004. Particle swarm optimization algorithm for permutation flowshop sequencing problem. Lecture Notes In Computer Science 3172, 382–389.

Trelea, I.C., 2003. The particle swarm optimization algorithm: convergence analysis and parameter selection. Information Processing Letters 85, 317–325.

Verma, A.K., Tamhankar, M.T., 1997. Reliability-based optimal task-allocation in distributed-database management systems. IEEE Transactions on Reliability 46, 452–459.

Vidyarthi, D.P., Tripathi, A.K., 2001. Maximizing reliability of distributed computing system with task allocation using simple genetic algorithm. Journal of Systems Architecture 47, 549–554.

Xia, W.J., Wu, Z.M., 2005. An effective hybrid optimization approach for multi-objective flexible job-shop scheduling problems. Computers and Industrial Engineering 48, 409–425.

Zhang, H., Li, X., Li, H., Huang, F., 2005. Particle swarm optimization-based schemes for resource-constrained project scheduling. Automation in Construction 14, 393–404.

Zhang, H., Li, H., Tam, C.M., 2006. Permutation-based particle swarm optimization for resource-constrained project scheduling. Journal of Computing in Civil Engineering 20, 141–149.