Available online at www.sciencedirect.com
SCIENCE DIRECT®
Parallel Computing 31 (2005) 839–857
PARALLEL COMPUTING
www.elsevier.com/locate/parco

Development of a parallel optimization method based on genetic simulated annealing algorithm

Z.G. Wang *, Y.S. Wong, M. Rahman

Department of Mechanical Engineering, National University of Singapore, 10 Kent Ridge Crescent, Singapore 119260, Singapore

Received 22 June 2004; received in revised form 10 March 2005; accepted 26 March 2005
Available online 13 June 2005

Abstract

This paper presents a parallel genetic simulated annealing (PGSA) algorithm that has been developed and applied to optimize continuous problems. In PGSA, the entire population is divided into sub-populations, and in each sub-population the algorithm uses the local search ability of simulated annealing after crossover and mutation. The best individuals of each sub-population are migrated to neighboring ones after a certain number of epochs. An implementation of the algorithm is discussed and the performance is evaluated against a standard set of test functions. PGSA shows some remarkable improvement in comparison with the conventional parallel genetic algorithm and the breeder genetic algorithm (BGA).

Keywords: Genetic algorithm; Simulated annealing; Parallel genetic algorithm

1. Introduction

Genetic algorithm (GA) was developed by Holland [1] and it is an adaptive global search method that mimics the metaphor of natural biological evolution. GA

* Corresponding author. Tel.: +65 68744644; fax: +65 67791459.
E-mail address: wangzhigang@nus.edu.sg (Z.G. Wang).

0167-8191/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.parco.2005.03.006

operates on a population of potential solutions by applying the principle of survival of the fittest to achieve an optimal solution. Owing to its ability to achieve the global or near global optimum, this algorithm has been applied to a large number of combinatorial optimization problems. In a GA approach to solve combinatorial optimization problems, a population of candidate solutions is maintained. To generate a new population, candidate solutions are randomly paired. For each pair of solutions, a crossover operator is first applied with a moderate probability (crossover rate) to generate two new solutions. Each new solution is then modified using a mutation operator with a small probability (mutation rate). The resulting two new solutions replace their parents from the old population to form a temporary new population. Every solution in the temporary population is ranked against other solutions based on a fitness criterion. The selection process is then used to determine a new population identical in size to the previous population, such that higher ranked candidates are allowed to have higher priority in the new population. GA iterates over a large number of generations until the termination criteria have been fulfilled. The pseudo code in Fig. 1 gives an overview of a traditional GA, where $P(t)$ is the population of individuals of generation t.

The successful application of GA depends on the population size or the diversity of individual solutions in the search space. If GA cannot hold its diversity well before the global optimum is reached, it may prematurely converge to a local optimum. Although maintaining diversity is the predominant concern of GA, it also reduces the performance of GA. Thus, various techniques have been attempted to find a trade-off between the population diversity and the performance of GA (exploration and exploitation).

An alternative approach is to combine GA with other optimization techniques, such as simulated annealing (SA). SA is a general-purpose stochastic optimization method that has proven to be quite effective in finding the global optima for many different NP-hard combinatorial problems. This paper presents a new hybrid of GA and SA, referred to as genetic simulated annealing (GSA). GSA is also parallelized to improve its computation performance further. The paper aims to demonstrate that parallel GSA is a powerful optimization strategy that overcomes the inherent weaknesses of conventional GA and SA.

```
1: t = 0
2: initialize P(t)
3: evaluate P(t)
4: while not termination-condition do
5:     t = t + 1
6:     select P(t) from P(t − 1)
7:     reproduce P(t)
8:     mutate P(t)
9:     evaluate P(t)
10: end while
```

Fig. 1. Pseudo code of the conventional genetic algorithm.

2. Hybrid of genetic algorithm and simulated annealing

GA and SA are both independently valid approaches toward problem solving with certain strengths and weaknesses. GA can begin with a population of solutions in parallel, but it suffers from poor convergence properties. By contrast, SA has better convergence properties if the starting temperature is sufficiently high and the temperature cooling rate is low. However, the higher temperature and the lower cooling rate reduce the performance of SA. In addition, parallelism cannot be easily exploited in SA.

Recently several researchers tried to combine GA and SA to provide a more powerful optimization method that has both good convergence control and efficient parallelization. Chen and Flann [2] had shown that the hybrid of GA and SA can perform better for ten difficult optimization problems than either GA or SA independently. Sirag and Weisser [3] proposed a unified thermodynamic genetic operator to solve ordering problems. The unified operator is applied to the conventional GA operation of crossover and mutation to yield offspring. This operator can ensure greater population diversity at high temperature and less population diversity at low temperature. Mahfoud and Goldberg [4] also introduced a GA and SA hybrid. Their hybrid runs SA procedures in parallel, which uses mutation as the SA neighborhood operator and incorporates crossover to reconcile solutions across the processors. A similar hybrid method of GA and SA was also used by Varanelli and Cohoon [5]. In addition, Chen et al. [6] also proposed a hybrid method, which maintains one solution per Processing Element (PE). Each PE accepts a visiting solution from other PEs for crossover and mutation. For the selection process, the SA cooling schedule and system temperature were used to decide whether the new generated individual was accepted or not. In this method, they used the local selection of SA to replace the conventional selection process of GA. Recently, Hiroyasu et al. [7] proposed an algorithm involving several processes. In each process SA is employed. The genetic crossover is used to exchange information between individuals at fixed intervals. Based on parallel simulated annealing in [4,7], Baydar [8] developed a parallel simulated annealing algorithm which also uses the survival of the fittest method, and acceptable results were found with this algorithm.

3. Genetic simulated annealing and parallel genetic simulated annealing

3.1. Genetic simulated annealing

Each of the above approaches to hybridize GA and SA described in Section 2 has its own strengths, because some good characteristics of GA and SA are maintained when combining GA and SA together. In this paper, a new GA and SA hybrid, GSA, is presented.

After crossover and mutation for a couple of individuals, there are four chromosomes: two parents and two offspring. In conventional GA, the two parents are replaced by their offspring. But in GSA, two chromosomes are chosen to form the next

generation from these four individuals. The selection criterion is based on the fitness values of these four individuals. Individuals with higher fitness values have a greater probability of surviving into the next generation. Those with less fitness values are not necessarily discarded. Instead, a local selection strategy of SA is applied to select them with a probability related to the current temperature (as in simulated annealing). In this selection process, a Markov chain is executed, which is composed of two offspring. Four parameters (f_{best}, f_{worst}, T_t, f_i) are involved to describe this selection process:

f_{best} the best fitness value of two parents;
f_{worst} the worst fitness value of two parents;
f_i the fitness value of one offspring ($i = 1, 2$);
T_t controlling temperature;

During the course of the Markov chain at temperature T_t, the fitness value f_i ($i = 1, 2$) of the trial chromosome is compared with f_{worst}. Chromosome i is accepted to replace the worst individual, if the following requirement is met:

$$\min\left\{1, e^{\frac{f_i - f_{\text{worst}}}{T_t}}\right\} \geqslant r$$

where r is a randomly generated number between 0 and 1. If chromosome i is accepted, the worst chromosome and the best one are updated then the course of the Markov chain continues again until the course finishes. After the implementation of the Markov chain, the best and the worst individuals are survived to the next generation.

Initially, the mutation probability p_m of GSA is set to a higher value, and a simple annealing process is then used to adjust p_m. After every ten generations, p_m is updated with $p_m \times \alpha$ until it reaches to a certain value, where α is the cooling rate of SA. Thus, at the initial stage, when manipulating the cooling schedule of SA properly, the initial higher temperature can ensure that the parents will be replaced by their offspring after mutation and crossover, whether they are much fitter or not. More importantly, the initial higher mutation probability is capable of improving population diversity significantly, which can eliminate the premature convergence problem of conventional GA. On the contrary, at the later stage the mutation probability and the temperature become lower, and the chances for the replacement of the fitter parents decrease greatly. In this way, the current best individuals always remain in the next generation. Thus, the possibility of removing potentially useful individuals in the last generation can be reduced because of the mutation operation. The pseudo-code of GSA is illustrated in Fig. 2, where $P(t)$ is the population of individuals of generation t, and n is the length of a chromosome.

In addition, good parallelizable property of GA is applied to parallelize GSA to improve its efficiency further, which will be explained in Sections 3.2 and 3.3. Thus, GSA shows tighter coupling of GA and SA, as SA controls a number of distinct GAs running in parallel.

Based on above analysis, GSA can maintain the strengths of both GA and SA. However, for problems with a large search space and costly function evaluations,

```
1: t = 0
2: initialize P(t) and temperature T_i
3: evaluate P(t)
4: while not termination-condition do
5:    t = t + 1
6:    select P(t) from P(t − 1)
7:    select individuals for reproduction from P(t)
8:    repeat
9:       select two unused individuals P_1, P_2
10:      crossover and mutation; generate two children C_1,C_2
11:      evaluate C_1, C_2
12:      for all i = 0 to 2 do
13:         if min[1, exp((f_i − f_worst)/T_t)] > random(0, 1) then
14:            accept C_i and replace the corresponding parent
15:            update the new best and worst points
16:         end if
17:      end for
18:   until all selected parents finish reproduction
19:   T_{t+1} = T_t × α; 0 < α < 1
20:   if the modulus of t divided by 10 == 0&&p_m > 1/n then
21:      p_m = p_m × α
22:   end if
23: end while
```

Fig. 2. Algorithm of genetic simulated annealing.

a faster GSA is needed. Here, a parallel version of GSA (PGSA) is presented to improve its efficiency.

3.2. Parallel genetic simulated annealing

There are several ways to parallelize GA [9], and the parallel GA (PGA) has been developed and successfully applied to optimize practical problems [10]. According to the nature of the population structure and recombination mechanisms used, PGA can be classified into four categories: single-population master–slave PGA, coarse-grained PGA, fine-grained PGA and hierarchical hybrids [11]. In single-population master–slave GA as shown in Fig. 3 (a), there is a single population, and the evaluation of fitness is distributed among several processors. The fine-grained parallel GA as shown in Fig. 3(b) treats each individual as a separate breeding unit; and the individuals may mate with those selected from a small local neighborhood. Since the neighborhoods overlap, fit individuals will migrate through the population. The coarse-grained parallel GA as shown in Fig. 3(c) is very popular and widely used. In a coarse-grained PGA the entire population is divided into several sub-populations. Each sub-population runs a conventional GA independently and concurrently on its own sub-population. After several epochs, best individuals migrate from one sub-population to another according to a migration topology. The hierarchical

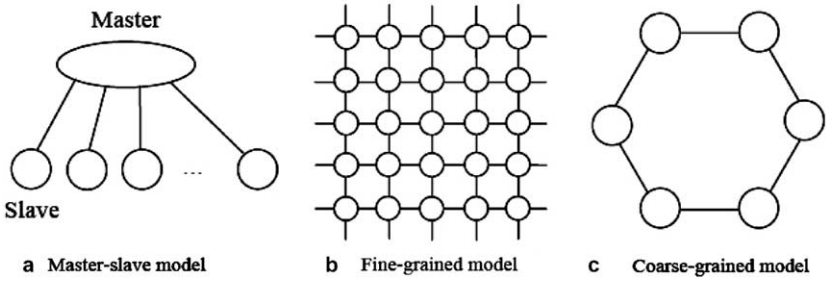

Fig. 3. Different models of parallel genetic algorithms.

parallel GA combines coarse-grained parallel GA with master–slave or fine-grained GA, so that it has the benefits of its components [11]. In this paper, the idea to parallelize GA has been borrowed to implement the parallelization of GSA. A master–slave/coarse-grained parallel GA, which combines both master–slave PGA and coarse-grained PGA, has been used to parallelize GSA.

In the master–slave/coarse-grained PGSA, the host program runs on the master processor, which decides on the global termination criterion. The whole population is equally divided into several sub-populations among the slave processors. Each slave processor runs a sequential GSA independently within its own sub-population on one processor. The pseudo code of PGSA is shown in Fig. 4, where $P(t)$ is the population of individuals of generation t, *myrank* is the rank of the process, $slns_{\text{migrate}}$, $slns_{\text{recv}}$ and $slns_{\text{delete}}$ are the migrants, received individuals and the individuals to be replaced, respectively. If the migration conditions are satisfied, each process, such as the source process, implements the function neighbor to find the destination processes according to the migration topology. The migrant individuals ($slns_{\text{migrate}}$) are selected and sent to the destination processes. After the migrant individuals ($slns_{\text{recv}}$) are received on the destination process, the individuals to be deleted ($slns_{\text{delete}}$) are

```
1: t = 0
2: initialize P(t)
3: evaluate P(t)
4: while not termination-condition do
5:    reproduction process of GSA
6:    if migration-condition satifies then
7:       dest = neighbor(myrank)
8:       slns_migrate = migrant_individual(P(t))
9:       send_string(dest, x_migrate)
10:      slns_recv = recv_string()
11:      slns_delete = delete_individual(P(t))
12:      replace_string(x_delete, x_recv, P(t))
13:   end if
14: end while
```

Fig. 4. Pseudo code of parallel genetic simulated annealing.

determined and replaced by received individuals ($slns_{recv}$). The same program is executed on each processor, but on different data (their own population) until the global optimum is achieved.

Although PGSA is related to the parallel hybrid method developed by Chen et al. [6], there are some important differences between PGSA in this paper and Chen's PGSA (C-PGSA). In C-PGSA, each PE maintains one solution, and each PE accepts a visiting solution from other PEs for crossover and mutation. In PGSA, each processor maintains its own sub-population of solutions and different processes exchange their best solutions after a certain number of epochs. Thus, the communication overhead between processors is much smaller. In C-PGSA, a normal SA-type probabilistic selection procedure is used to retain the proof of convergence of SA. In PGSA, a Markov chain is used to realize the local selection of SA, which can improve the selection performance of SA.

3.3. Implementation details of parallel genetic simulated annealing

3.3.1. Representation

PGSA uses a real-value coding scheme to represent the chromosome, and each chromosome vector is coded as a vector of real-value point numbers of the same length as the solution vector. Let $x = (x_1, x_2, \ldots, x_n)$ be the encoding of a solution, where $x_i (\in R)$ represents the value of the ith gene in the chromosome x. Initially, x_i is selected within the desired domain, and reproduction operators of GA are carefully designed to preserve this constraint.

3.3.2. Selection

Based on their fitness function values, individuals are appropriately selected for recombination. The first step is to assign fitness values to all individuals according to their values of objective function(s). Sometimes the fitness value needs to be scaled for further use. Scaling is important to avoid early convergence caused by dominant effect of a few strong candidates in the beginning, and to differentiate relative fitness of candidates when they have very close fitness values near the end of run [12]. In GA, there are mainly three selection procedures: proportional selection, tournament selection and rank-based selection [13]. The proportional approach is usually called "roulette wheel" selection. Fitness values of individuals represent the width of slots of the wheel, and selection is based on the slot widths of individuals [14]. Individuals with larger slot widths will have a higher probability to be selected. In the tournament approach, a sub-group is initially selected randomly from the population with or without replacement. Then, a "tournament" competition is hold in this sub-group, and the winner is inserted into the next population. This process is repeated until the new population is formed [13]. For the rank-based approach, the individuals are sorted based on their fitness values. The rank N is assigned to the best solution, and the rank 1 is assigned to the worst one. Based on their ranks, the selection probability is then assigned to the individuals. Rank-based selection behaves in a more robust manner than the proportional one. Therefore in this paper, the rank-based fitness selection is used.

3.3.3. Crossover and mutation

The basic operators for producing new individuals in GA are crossover and mutation. *Crossover* may produce better individuals that have some genetic material of both parents. The conventional crossover operator combines sub-strings belonging to their parents. For real-value encoding, this type of crossover does not change the value of each variable; so it cannot perform the search with respect to each variable. Therefore, it is not suitable in this study and consequently, a modified crossover operator, called convex recombination [15], is used. It operates as follows.

Consider that the crossover takes place at position i, let $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$ be the parents strings. Then, the offspring u and v are of the form:

$$u_i = ax_i + (1 - a)y_i, \quad i = 1, \ldots, n,$$
$$v_i = ay_i + (1 - a)x_i, \quad i = 1, \ldots, n.$$

where a is a random number in the interval $[0, 1]$. In this paper, the two-point crossover is applied to each couple of individuals, whose string length is greater than two. And for individuals with a string length of two, one-point crossover is used.

Mutation simply changes the value for a particular gene with a certain probability. It helps to maintain the vast diversity of the population and also prevents the population from stagnating. However, at later stages, it increases the probability that good solutions will be destroyed. Normally, the probability that mutation will occur is set to a low value (e.g., 0.01) so that accumulated good candidates will not be destroyed. In this study, the local selection of SA is applied after mutation, such that at the later stage, only better solutions are retained after mutation; so the initial value of mutation probability can be larger than the recommended values in [16].

For the real-value coding scheme, different mutation operators can be used, such as uniformly distributed mutation, Gaussian mutation, range mutation and non-uniform mutation. The first two mutation methods have been used in this study. In the uniformly distributed mutation, the mutation operator randomly chooses a number z in the interval defined by $[-A, A]$, where A is called the mutation range. The new point is given by: $x_m = x + z$ [17]. In Gaussian mutation, a random value z is chosen from a normal Gaussian distribution $N(0, \sigma)$, where σ is the standard deviation [18]. Uniformly distributed mutation is more commonly used for searches in a large region. The Gaussian mutation performs better searches in a small local area [15]. In this study, at the initial stage, uniformly distributed mutation is used. When the decreasing rate of the average fitness values is less than 0.01, Gaussian mutation is used.

3.3.4. Migration policy, rate, topology and frequency

In the implementation of the coarse-grained PGSA, some parameters of concern are: migration policy that determines the selection of individuals to migrate, migration rate or the number of individuals to migrate, the frequency of migration, and the migration topology.

Two methods to choose the individual to migrate are often attempted: elitist strategy and probabilistic tournament selection. In the first method, the best individuals are sent to a neighboring sub-population. In the second, the fitter individual is

selected via a probabilistic binary tournament with a certain probability. Gordon and Whitley [9] compared these two methods and indicated that the elitist strategy performed better than the tournament selection. Thus, the elitist strategy is used to choose the migration individuals. The top 1% of the best individuals are migrated to replace the worst individuals of other sub-populations. If 1% of population size is not an integer, it will be rounded off to the next integer that is greater than its fractional value. Based on the ladder structure concept in [19], a similar ladder neighborhood relation is used to implement PGSA, and the 8-processor structure of this relation is shown in Fig. 5(a), which shows that every slave processor is connected with other four slave processors. Furthermore, another topology is proposed as shown in Fig. 5(b). Every processor except the master processor has five neighboring processors. The advantage of this topology is that it can spread the migrant individuals quickly among the slave processors. The frequency of migrating individuals is given in Section 4.1.

3.3.5. Termination criterion

In PGSA, the host program, which runs on the master processor, decides on the global termination criterion. After certain iterations, each slave processor sends its top 1% of the best individuals to the master processor. After receiving the best individuals from all of the slave processors, the termination flag of PGSA is set on the master processor, if the following criterion is fulfilled:

$$| f^k_{\text{best}} - f^{k-\Delta k}_{\text{best}} | \leqslant \varepsilon \, | f^k_{\text{best}} | \tag{1}$$

$$\text{or} \quad | f^k_{\text{best}} - f_{\text{best}} | \leqslant \varepsilon \tag{2}$$

where f^k_{best} is the best fitness value of an individual on the master processor at generation k, and the termination criterion is checked at every Δk generations; f_{best} is the global minimum and ε is a constant value 10^{-3}. For F1–F7 and F9, inequality (1) was used as the termination criterion. For F8, inequality (2) was chosen as the

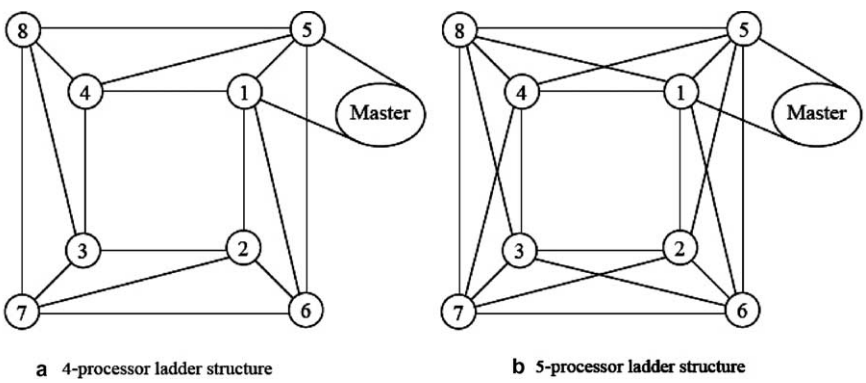

Fig. 5. Schematic diagram of the implementation of PGSA.

termination criterion, where $f_{best} = 0$. After setting the termination flag, the master processor sends it to each slave processor. According to the termination flag, each slave processor decides whether GSA continues to run or not.

4. Numerical results and discussion

A set of test functions from available literatures [19–21] has been used to compare the performance of PGSA with other algorithms. This is shown in Table 1. De Jongs suite of functions F1–F5 [20] is extensively used in the GA community. Functions F6–F8 are also widely used to test the performance of global search algorithms. All tests were run on SUN Workstation network, which consists of 42 SUN Blade 2000 workstations with a fast Ethernet interconnect (100 MBytes/s). The hardware and software details of Sun Blade 2000 are as follows: Ultra-SPARC III Cu

Table 1
A set of standard test function

Function	Function equation	Parameters intervals
F1	$f_1(x) = \sum_{i=1}^{3} x_i^2$	$-5.12 \leqslant x_i \leqslant 5.12$, $i = 1,2,3$
F2	$f_2(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$	$-2.048 \leqslant x_i \leqslant 2.048$, $i = 1,2$
F3	$f_3(x) = \sum_{i=1}^{5} \text{integer}(x_i)$	$-5.12 \leqslant x_i \leqslant 5.12$, $i = 1,2,\ldots,5$
F4	$f_4(x) = \sum_{i=1}^{30} ix_i^4 + \text{Gauss}(0,1)$	$-1.28 \leqslant x_i \leqslant 1.28$, $i = 1,2,\ldots,n$
F5	$f_5(x) = 0.002 + \sum_{j=1}^{25} \frac{1}{j + \sum_{i=1}^{2}(x_i - a_{ij})^6}$	$-65.536 \leqslant x_i \leqslant 65.536$, $i = 1,2$
F6	$f_6(x) = nA + \sum_{i=1}^{n}[x_i^2 - A\cos(2\pi x_i)]$	$-5.12 \leqslant x_i \leqslant 5.12$, $i = 1,2,\ldots,n$
F7	$f_7(x) = \sum_{i=1}^{n}[-x_i \sin(\sqrt{\lvert x_i \rvert})]$	$-500 \leqslant x_i \leqslant 500$, $i = 1,2,\ldots,n$
F8	$f_8(x) = \sum_{i=1}^{n} \frac{-x_i^2}{4000} - \prod_{i=1}^{n} \cos(\frac{x_i}{\sqrt{i}}) + 1$	$-600 \leqslant x_i \leqslant 600$, $i = 1,2,\ldots,n$
F9	$f_9(x) = \sum_{i=1}^{n}[100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2]$	$-5.12 \leqslant x_i \leqslant 5.12$, $i = 1,2,\ldots,n$

900 MHz processor, 2 GBytes memory, swap memory 4 GBytes, and Solaris 8 operating system.

An extensive performance evaluation for functions F1–F8 has been done for the PGA by Mühlenbein et al. [19]. In their investigation, the efficiency of the search method was demonstrated by varying the problem size n. Mühlenbein and Schlierkamp-Voosen [17] increased the problem size n further to investigate the advantage of the scaling of their search method. The number of function evaluations was defined as efficiency criteria in [17,19]. For comparison, in this section, the efficiency of search methods is also defined by the number of function evaluations needed to obtain the optimal solutions. When the minimal value of each function on the master processor is reached, it will send the termination signal to all slave processors. After these processors have received the termination signal, they stop running and send the number of function evaluations done so far to the master processor to sum them up.

4.1. Experiment setup

Both GA and SA have several internal control parameters. Thus, PGSA, which is a hybrid of GA and SA, also has several control parameters, that can be described as follows:

$$\text{PGSA} = \{P_0, \lambda, \mu, \sigma, \delta, \tau, GSA, T_0, \alpha, t\} \tag{3}$$

where P_0 is the initial population, λ is the number of sub-populations, μ is the population size of each sub-population, σ is the migration interval in number of generations, δ is the number of neighbors, T_0 is the initial temperature, α is the cooling rate, and t is the termination criterion.

In order to show the robustness of PGSA, it is not necessary to tune all of these parameters to a specific function. Based on the recommended values from previous work [11,19], the parameters for F1–F9 are shown in Table 2, where p_m and p_c are the respective mutation and crossover probabilities. As it is much more difficult to get the global minimal value of Function F8, the larger population size and mutation rate were used. Furthermore in order to get more representative results, the average values for PGSA listed in the following tables were based on 50 runs across the test suite of functions.

Table 2
PGSA's parameters setting for Function F1–F9

Parameters	F1–F5	F6	F7	F8	F9
λ	8	8	8	16	20
μ	20	20	20	50	100
σ	10	10	10	20	20
p_m	0.5	0.1	0.1	0.3	0.05
p_c	0.65	0.65	0.65	0.65	0.85
T_0	200	200	200	200	800
α	0.85	0.85	0.85	0.85	0.85

4.2. Results and discussion for lower dimension problems

The computation results for F1–F9 using PGSA are shown in Table 3, where n is the problem size and *FE* indicates the number of function evaluations. For comparison, the average number of function evaluations with optimized PGA by Mühlenbein et al. [19] is also included in Table 3. For Functions F1–F5, the difference in performance between PGA and PGSA is very small. These problems could be too small for PGSA or PGA to work effectively. Comparison of more complex functions F6–F9 shows more significant difference.

Functions F6 and F7 with lower dimensions are also easily solved by PGSA and PGA. But in [19], the global optimum of F7 was not found in 4 of the 50 runs. In this study, PGSA found the optimum of F7 in all 50 runs. Because PGSA can maintain a good diversity with a higher mutation probability at the initial stage, it can eliminate premature convergence to sub-optimal minima. At the later stage, the local selection strategy of SA can ensure that best solutions are not discarded after crossover and mutation. Therefore, PGSA can approach or converge to the global minimum.

Griewank's function F8 is regarded as one of the most difficult test functions. Function F8 has its global minimum $f_{\text{best}} = 0$ at $x_k = 0$, and the local minima are located approximately at $x_k = m\pi\sqrt{k}$, where $k = 1,\ldots,n$, and m is any integer value. Four sub-optimal minima (≈ 0.0074) exist at $\vec{x} = (\pm\pi, \pm\pi\sqrt{2}, 0, \ldots, 0)$ in ten dimensions. The average number of function evaluations is 6600 by Griewank [21], but only one of the four sub-optima was found. An average of 59 520 evaluations is needed to solve this problem by Mühlenbein et al. [19], but they did not comment on their results. PGSA found the minimum values (<0.001) of F8 with only about half of the number of function evaluations using PGA. More importantly, in all 50 runs, these minimum values were found. Therefore, PGSA is able to obtain much better solutions with a higher convergence speed than PGA.

Function F9 with a dimension of 50 was extended from that of Rosenbrock. For this kind of functions, the mutation scheme is not efficient, and line recombination is able to leave the saddle point after a steep valley [22]. Line recombination uses components from both mutation and recombination, and it creates new points in

Table 3
Performance comparison between PGA [19] and PGSA

Function	n	PGA [19]		PGSA	
		λ	*FE*	λ	*FE*
F1	3	8	1526	8	1287
F2	2	8	1671	8	1473
F3	5	8	3634	8	1769
F4	30	8	5243	8	4025
F5	2	8	2076	8	1476
F6	20	8	9900	8	6705
F7	10	8	8699	8	6006
F8	10	16	59520	16	25690
F9	50	–	–	20	537235

a direction given by two parents' points. In this study, the line recombination was also tried for F9 instead of the crossover mentioned in Section 3.3.3. Even with the use of line recombination, PGSA still converges to suboptimal points in some cases. A revised Gaussian mutation, which has non-zero expectation, is able to help PGSA leave these sub-optimal points. Let $x = (x_1, \ldots, x_n)$ be a parent string, the new offspring u is of the form:

$$u_i = \begin{cases} x_i + N(0, \sigma), & i = 0 \\ x_i + N\left(\frac{x_{i-1} - x_i}{2}, \sigma\right), & 0 < i < n \end{cases} \tag{4}$$

where σ is the standard deviation. In our study, $\sigma = 0.005$. We choose $(x_{i-1} - x_i)/2$ as the expectation because objective function values depend on the difference of values of two consecutive genes. A typical run using PGSA and PGA with the revised Gaussian mutation and PGSA with normal Gaussian mutation is shown in Fig. 6.

From Fig. 6, it can be seen that the best evaluation values decrease slightly following a steep decrease initially for these three cases. PGSA and PGA with the revised Gaussian mutation can move out from such saddle points. However, it is very difficult for PGSA with normal Gaussian mutation to leave the saddle points. Even after 1000 generations, it still stagnates at the sub-optimal points, while PGSA and PGA with the revised Gaussian mutation can exploit the more optimal points gradually. Hence, the revised Gaussian mutation is more efficient than the normal one, because it uses information from its former consecutive genes to move the solutions towards more optimal points. On the contrary, normal Gaussian mutation just changes the value of a particular gene randomly, while the objective function value depends

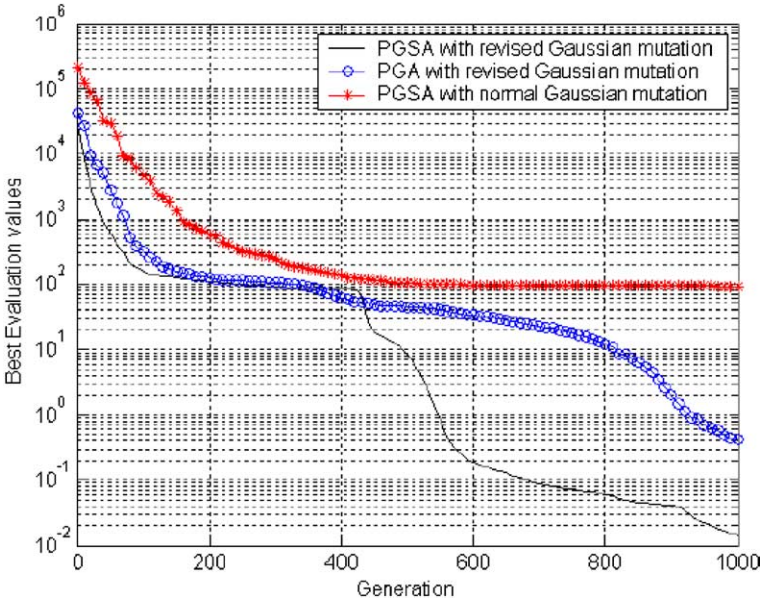

Fig. 6. Best function evaluation values for Rosenbrock function ($n = 50$).

not only on this particular gene, but also on the difference between its value and its former consecutive one. Thus, it cannot guarantee that it will move out of the sub-optimal points. However, when choosing $(x_{i-1} - x_i)/2$ as the expectation, the revised Gaussian mutation can solve this problem. In addition, with the help of the revised Gaussian mutation, the performance of PGSA outweighs that of PGA. The reason for this is discussed in the following section.

4.3. Results and discussion for higher dimension problems

4.3.1. Discussion of speed-up of PGSA

For Function F7, the sub-optima are far away from the optimization so that a parallel search with sub-populations seems to be promising. Function F7 with dimensions of 50 and 100 has been attempted, and the testing parameters settings are shown in Table 4. The measured computation time and communication time are shown in Figs. 7 and 8, which indicate a super-linear speedup when the number of processors increases from 6 to 10. In the test for F7, the entire population size is kept constant. When less number of processors is used, the population size in each processor is increased, resulting also in the increase in the population diversity. The increased population diversity in each processor may delay the convergence of PGSA. Therefore, the delay of convergence when less number of processors is used may result in super-linear speedup when a few more processors are used. Another possible reason for this super-linear speedup is that smaller population size

Table 4
PGSA's parameters setting for Function F7 ($n = 50$ and 100)

n	σ	p_m	p_c	T_0	α	$\lambda - \mu$							
50	30	0.05	0.65	400	0.85	6–134	8–100	10–80	12–68	14–58	16–50	18–44	20–40	22–36
100	50	0.05	0.65	600	0.85	6–334	10–200	14–142	16–126	18–112	20–100	22–92	24–84	26–78

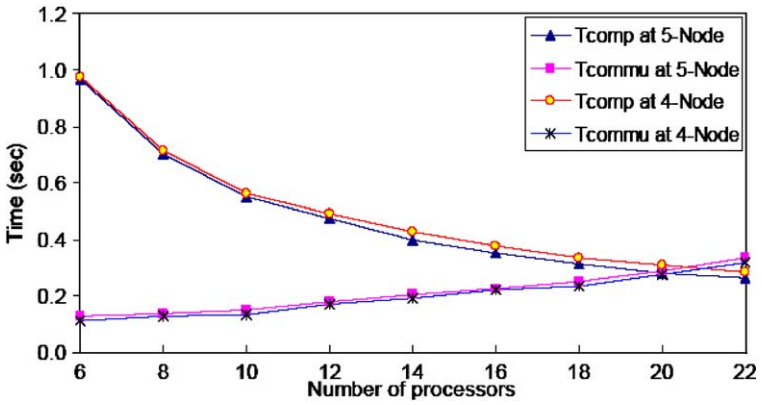

Fig. 7. Computation time and communication overhead for F7 ($n = 50$).

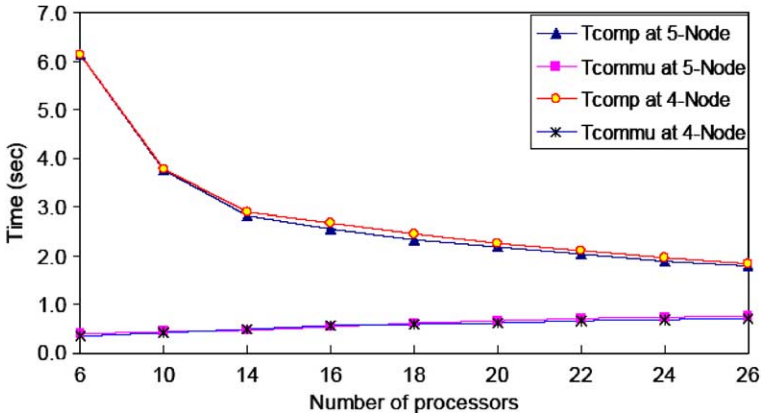

Fig. 8. Computation time and communication overhead for F7 ($n = 100$).

may fit well with the processors caches, and the reduction of memory access time can help to produce super-linear speedup. Similar results had also been obtained in [11]. However, when even more processors are involved in the computation, the decreasing rate of population diversity in each processor becomes smaller. So that the increasing rate of computation time becomes smaller, and at the same time the communication overhead is increased. For F7 (with $n = 50$), when 24 processors are used, the savings on computation costs cannot compensate for the rapidly increasing costs of communication.

When fewer processors are used for F7, there is only slight difference in computation time between 5-processor neighborhood ladder structure and the 4-processor one. However, when more processors are involved in the computation, the difference in computation time between these two types of ladder structure becomes obvious, as shown in Figs. 7 and 8. Although, with the 5-processor neighborhood ladder structure, the communication costs between processors are slightly higher than those with the 4-processor structure, the savings of computation time can compensate for the increased communication overhead. Therefore, the 5-processor structure has been chosen as the migration topology.

Similarly, based on the trade-off between communication cost and computation time, the optimal number of processors used in this study has been determined.

4.3.2. Computation results for F6 and F7 with higher dimension

Mühlenbein et al. [19] found the global minimum of F6 of dimension 400 and F7 of dimension 150 on a 64-processor computer using PGA. For comparison in this paper, F6 and F7 of much higher dimension have also been attempted using PGSA. The test parameters are listed in Table 5. The same termination criterion (as given in Section 3.3.5) was used in order to directly compare the efficiency of PGSA and PGA.

It is instructive to compare our results with the results from [19], as shown in Table 6. Function F6 with dimensions 500 and 1000 and Function F7 with dimensions 200 and 400 have also been investigated, which were not solved using

Table 5
PGSA's parameters setting for Function F6 and F7 with higher dimension

Parameters	Dimension of F6						Dimension of F7				
	50	100	200	400	500	1000	50	100	150	200	400
λ	8	16	16	16	16	32	16	16	16	16	32
μ	20	20	40	80	100	100	50	100	150	200	200
σ	20	20	40	40	50	50	30	50	50	50	50
p_m	0.05	0.05	0.05	0.65	0.80	0.80	0.05	0.05	0.05	0.05	0.05
p_c	0.65	0.65	0.65	0.65	0.65	0.65	0.65	0.65	0.65	0.65	0.65
T_0	400	500	1000	2000	2000	2000	400	600	800	1000	1200
α	0.85	0.85	0.85	0.85	0.85	0.85	0.85	0.85	0.85	0.85	0.85

Table 6
Performance comparison between PGA [19] and PGSA

Function	n	PGA [19]			PGSA		
		λ	μ	*FE*	λ	μ	*FE*
F6	$n = 50$	8	20	42753	8	20	18378
	$n = 100$	16	20	109072	16	20	77489
	$n = 200$	32	40	390768	16	40	257894
	$n = 400$	64	40	7964400	16	80	356333
	$n = 500$	–	–	–	16	100	565348
	$n = 1000$	–	–	–	32	100	1168424
F7	$n = 50$	32	20	119316	16	50	70853
	$n = 100$	64	20	1262228	16	100	273222
	$n = 150$	64	40	7041440	16	150	689888
	$n = 200$	–	–	–	16	200	1399284
	$n = 400$	–	–	–	32	200	2846835

PGA in [19]. It can be seen that the number of function evaluations using PGSA is much smaller than that using PGA. The performance of PGSA gets better with the higher problem size n.

A typical run is shown in Fig. 9. Initially, the difference in performance between PGSA and PGA is very small. After 100 generations, it becomes larger, and for PGA the individuals of the sub-population become similar to those of PGSA. In some cases, PGA even converges around sub-optimal minima. Thus, the average evaluation values are smaller than those of PGSA. However, for PGSA, at this stage, the smaller sub-optimal minima have been found. The difference between the average and the best evaluation values is larger than that of PGA, which means that more diverse individuals exist in the sub-populations. As the optimization process proceeds for PGSA, the controlling temperature gets lower, and the probability that the best individuals found are kept in the next generation becomes larger. When the temperature is very small, the best individuals are always retained in the next generation. Normally, the mutation probability is set to a low value (e.g., 0.01) for GA or PGA; otherwise the accumulated good candidates are destroyed at the later stage. However, this negative effect of mutation has been eliminated for PGSA, because the

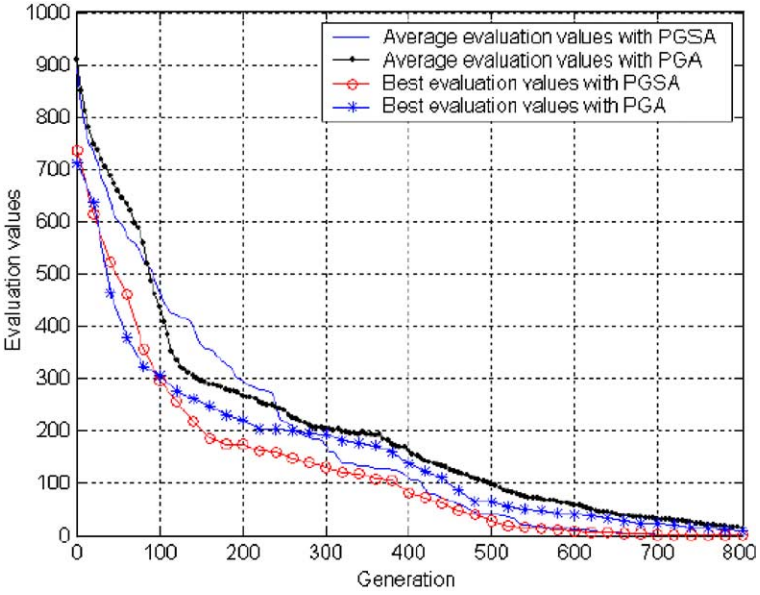

Fig. 9. Average and best function evaluation values for Rastrigin function ($n = 50$).

local selection of SA takes place during this later stage and only better individuals are retained after mutation. Finally, because of the maintained diversity and the local selection of SA, PGSA can converge to the optimal solution within much smaller number of generations. It can be seen that PGSA found the global optimal solution after 700 generations, whilst PGA still could not find the optimal solution even after 800 generations.

4.3.3. Computation results for F8 with higher dimension

Mühlenbein and Schlierkamp-Voosen [17] used breeder genetic algorithm (BGA) to optimize Functions F6 and F7 with higher dimensions. However, the number of function evaluations cannot be directly compared because a different termination criterion was used for BGA. In order to compare the efficiency of PGSA with that of BGA, the same termination criterion as that for BGA was used for Function F8. The computation results using BGA [17] and PGSA are listed in Table 7. The number of

Table 7
Performance comparison between BGA [17] and PGSA

Function	n	PGA [19]				PGSA			
		λ	μ	*FE*	$668n\ln(n)$	λ	μ	*FE*	$58690\sqrt{n}/\ln(n)$
F8	$n = 20$	1	500	66000	40023	14	50	*39707*	87614
	$n = 100$	1	500	361722	307625	16	100	*123610*	127444
	$n = 200$	1	500	748300	707855	16	100	*157568*	156654
	$n = 400$	1	500	1630000	1600920	16	100	*196054*	195912

function evaluations scales almost exactly with $n \cdot \ln(n)$ for Function F8 [17]. When the problem dimension size n is in the range of 100–400, the number of function evaluations scales exactly with $\sqrt{n}/\ln(n)$ in this study. With such scaling ability, the advantage of investigating the scaling of PGSA has been demonstrated. PGSA achieves much better performance with the higher problem size n. Thus the performance of PGSA is better in comparison to that of BGA.

5. Conclusions

In this paper, a new GA and SA hybrid (GSA) is firstly presented, which inherits the strengths of GA and SA and overcomes their weaknesses. The extended ideas of simulated annealing (SA) have been used in: (1) adjustment of the mutation rate; (2) the local selection of individuals which are retained in the temporary population after crossover and mutation. In GSA, at the initial stage, the higher mutation rate is helpful for maintaining population diversity. After crossover and mutation, the local selection of SA can ensure that good candidates still exist in the next generation at the later stage. Therefore by maintaining more diverse sub-populations at the initial stage, GSA mitigates the premature convergence of the standard GA. On the other hand, at the later stage, local selection strategy of SA ensures that increasing number of good candidates exists in the next generation. It can narrow the search space so that fast convergence can be achieved. Then PGSA is described by implementing the parallelization of GSA, which improves the efficiency of GSA further.

In this study, nine different functions have been tried with the proposed algorithm. The main difference between them lies in the objective function. To use the proposed algorithm for new problems, it is only necessary to incorporate the objective function into the program and change some parameters if necessary. Thus, the programmability of the algorithm is good. The numerical results show that PGSA has faster convergence to global optimum solution than PGA, and thus, is superior to the conventional PGA. In comparison to the other advanced search method, such as breeder genetic algorithm (BGA) [17] using the same termination criterion, the performance of PGSA is better. More importantly, PGSA performs better with the larger problem size.

References

[1] J.H. Holland, Adaptation in Natural and Artificial Systems, University of Michigan Press, Ann Arbor, MI, 1975.

[2] H. Chen, N. Flann, Parallel simulated annealing and genetic algorithms: a space of hybrid methods, in: Proceedings of International Conference on Evolutionary Computation—PPSN III, in: Y. Davidor, H.P. Schwefel, R. Manner (Eds.), Lecture Notes in Computer Science, vol. 866, Springer-Verlag, Berlin, 1994, pp. 428–438.

[3] D.J. Sirag, P.T. Weisser, Toward a unified thermodynamic genetic operator, in: J.J. Grefenstette (Ed.), Proceedings of Second International Conference on Genetic Algorithms, Lawrence Erlbaum Associates, Hillsdale, NJ, 1987, pp. 116–122.

[4] S.W. Mahfoud, D.E. Goldberg, Parallel recombinative simulated annealing: a genetic algorithm, Parallel Computing 21 (1995) 1–28.
[5] J.V. Varanelli, J.C. Cohoon, Population-oriented simulated annealing: a genetic/thermodynamic hybrid approach to optimization, in: L.J. Eshelman (Ed.), Proceedings of Sixth International Conference on Genetic Algorithms, M. Kaufman, San Francisco, Calif., 1995, pp. 174–181.
[6] H. Chen, N.S. Flann, D.W. Watson, Parallel genetic simulated annealing: a massively parallel SIMD algorithm, IEEE Transactions on Parallel and Distributed Systems 9 (1998) 126–136.
[7] T., Hiroyasu, M. Miki, M. Ogura, Parallel simulated annealing using genetic crossover, in: Proceedings of the IASTED International Conference on Parallel and Distributed Computing Systems, Las Vegas, 2000, pp. 145–150.
[8] C. Baydar, A hybrid parallel simulated annealing algorithm to optimize store performance, Workshop on GECCO 2002, 2002.
[9] V.S. Gordon, D. Whitley, Serial and parallel genetic algorithms as function optimizers, in: S. Forrest (Ed.), Proceedings of Fifth International Conference on Genetic Algorithms, M. Kaufmann, San Mateo, CA, 1993, pp. 177–183.
[10] E. Alba, J.M. Troya, A survey of parallel distributed genetic algorithms, Complexity 4 (1999) 31–52.
[11] E. Cantu-Paz, Efficient and Accurate Parallel Genetic Algorithms, Kluwer Academic Publishers, Boston, 2000.
[12] D.E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning, Addison CWesley, 1989.
[13] T. Blickle, L. Thiele, A comparison of selection schemes used in evolutionary algorithms, Evolutionary Computation 4 (1996) 361–394.
[14] D.T. Pham, D. Karaboga, Intelligent Optimisation Techniques: Genetic Algorithms, Tabu Search, Simulated Annealing and Neural Networks, Springer, New York, 2000.
[15] D. Dumitrescu, B. Lazzerini, L.C. Jain, A. Dumitrescu, Evolutionary Computation, CRC Press, Boca Raton, FL, 2000.
[16] J. Hesser, R. Männer, Towards an optimal mutation probability for genetic algorithms, in: Proceedings of Parallel Problem Solving from Nature: 1st Workshop, PPSN I, in: H.-P. Schwefel, R. Männer (Eds.), Lecture Notes in Computer Science, vol. 496, Springer-Verlag, Berlin, 1991, pp. 23–32.
[17] H. Mühlenbein, D. Schlierkamp-Voosen, Predictive models for the breeder genetic algorithm: I. Continuous parameter optimization, Evolutionary Computation 1 (1993) 25–49.
[18] Th. Bäck, F. Hoffmeister, H.-P. Schwefel, A survey of evolution strategies, in: R.K. Belew, L.B. Booker (Eds.), Proceedings of Fourth International Conference on Genetic Algorithms, M. Kaufmann, San Mateo, CA, 1991, pp. 2–9.
[19] H. Mühlenbein, M. Schomisch, J. Born, The parallel genetic algorithm as function optimizer, Parallel Computing 17 (1991) 619–632.
[20] K.A. De Jong, An analysis of the behavior of a class of genetic adaptive systems, Ph.D. Thesis, University of Michigan, 1975.
[21] A.O. Griewank, Generalized descent for global optimization, Journal of Optimization Theory and Applications 34 (1981) 11–39.
[22] D. Schlierkamp-Voosen, H. Mühlenbein, Strategy adaptation by competing subpopulations, in: Proceedings of International Conference on Evolutionary Computation—PPSN III, in: Y. Davidor, H.P. Schwefel, R. Manner (Eds.), Lecture Notes in Computer Science, vol. 866, Springer-Verlag, Berlin, 1994, pp. 199–208.