J Glob Optim (2007) 39:261–289
DOI 10.1007/s10898-007-9138-0

ORIGINAL PAPER

Parallelizing simulated annealing algorithms based on high-performance computer

Ding-Jun Chen · Chung-Yeol Lee · Cheol-Hoon Park · Pedro Mendes

Received: 28 July 2005 / Accepted: 12 January 2007 / Published online: 20 February 2007

Abstract We implemented five conversions of simulated annealing (SA) algorithm from sequential-to-parallel forms on high-performance computers and applied them to a set of standard function optimization problems in order to test their performances. According to the experimental results, we eventually found that the traditional approach to parallelizing simulated annealing, namely, parallelizing moves in sequential SA, difficultly handled very difficult problem instances. Divide-and-conquer decomposition strategy used in a search space sometimes might find the global optimum function value, but it frequently resulted in great time cost if the random search space was considerably expanded. The most effective way we found in identifying the global optimum solution is to introduce genetic algorithm (GA) and build a highly hybrid GA+SA algorithm. In this approach, GA has been applied to each cooling temperature stage. Additionally, the performance analyses of the best algorithm among the five implemented algorithms have been done on the IBM Beowulf PCs Cluster and some comparisons have been made with some recent global optimization algorithms in terms of the number of functional evaluations needed to obtain a global minimum, success rate and solution quality.

Keywords Simulated annealing · Genetic algorithms · Parallel and distributed processing · Message-passing interface (MPI) · High-performance computing

D.-J. Chen (✉) · C.-Y. Lee · C.-H. Park
Department of Electrical Engineering and Computer Science, Korea Advanced Institute of Science and Technology, Daejeon 305-701, Republic of Korea
e-mail: chen_dingjun@yahoo.com

P. Mendes
Virginia Bioinformatics Institute, Virginia Tech, Blacksburg, VA 24060, USA

1 Introduction

Many application problems in engineering are formulated as optimization problems; for example, in diverse disciplines such as physics, chemistry and molecular biology, the optimization problem min $f(\vec{x})$, subject to only lower- and upper-bound constraints on the variables, is frequently met. In order to solve such a type of optimization problems, a large number of optimization methods have been developed. Some of deterministic methods apply deterministic heuristics, such as modifying the search trajectory in trajectory methods and adding penalties in penalty-based methods, to bring a search out of a local minimum [1]. In general, these methods do not work well if the search space is too large or the optimization function has many distinct local minima. In the last 20 years, numerous stochastic optimization algorithms have emerged as a powerful and broad method for solving optimization problems. These probabilistic global optimization methods rely on probability to make decisions. In general, they have high computational demand. Simulated annealing is one of the most popular stochastic global optimization methods and has been successfully applied to solve many nonlinear optimization problems [2–6]. Up to today, many novel extensions of the simulated annealing method have been proposed for optimizing difficult functions, such as Fast Simulated Annealing and Very Fast Simulated Re-annealing (VFSR) [2–4]. Those methods mainly focus on finding a proper temperature annealing schedule so as to make SA converge to the lowest functional minimum (namely, the global minimum) as quickly as possible. Some researchers have ever tried to introduce the parallelism in SA. Actually, due to the fact that SA is a naturally sequential algorithm, it is difficult to parallelize SA without changing its serial nature. In the last decade, methods to efficiently parallelize SA have always been a controversial issue and were discussed by Ellen et al. [7], Hamma et al. [8], Ingber [5], Yong et al. [9] and Chen et al. [10]. The latest five parallel SA algorithms developed by Esin and Linet [11] are implemented for the Distributed Memory MIMD systems and based upon 8 Pentium II 350 MHZ PCs connected by 10Mbit/second Ethernet using the Parallel Virtual Machine (PVM) Linux programming environment. The study based on such a computing platform impossibly offers us comprehensive performance analyses, we think. The main reason is that we actually do not know some performances how to vary if the parallel SA runs with over 8 PCs, such as the scalability of the parallel SA.

With rapid advances in personal computers and networks in the last decade, it is possible to use SA to find quickly a global optimum with the lowest time-consuming cost by converting existing simulated annealing software from sequential-to-parallel forms on a high-performance computer. This approach is becoming more appealing to us because PCs clusters are more economical than supercomputers and the developed parallel software is highly portable. Certainly, it is unrealistic to expect to find one general parallel SA approach to work well for every kind of nonlinear optimization problem. We think, however, we may attempt to find such a parallel stochastic optimization approach that fits many more problems of the type we are solving. Thus, in order to thoroughly investigate the performance of parallel approaches for SA on a high-performance computer and to try to find a as robust and good parallel SA as possible, we have carried out five different conversions of simulated annealing software from sequential-to-parallel forms on IBM Beowulf PCs xSeries clusters using Message-Passing Interface (MPI) and applied them to a set of standard multimodal functions for evaluating their performances.

The organization of this paper is as follows: In Sect. 2, five strategies are used and detailed implementations on how to convert the existing sequential simulated annealing software from sequential-to-parallel forms are presented. The computational results, detailed performance analyses of algorithms and discussions are given in Sect. 3. Some conclusions are given in Sect. 4. Test functions used for testing the proposed parallel optimization algorithms are given in Appendix A.

2 Parallelizing simulated annealing

2.1 Strategies used for parallelizing SA

In this paper, we consider applying SA only to function minimization problems. The duality principle suggests that we can convert a maximization problem into a minimization one by multiplying the objective function by −1. Therefore, sequential SA generally may work as follows: At the beginning, SA randomly generates a single point, and then begins to perform the following iterative operations. A neighbor is created from a current solution by changing a part of the solution and if the neighbor's cost function (or "energy") is less than that of the existing solution, the neighbor replaces the current solution. Otherwise, the neighbor is accepted probabilistically. The probability of acceptance depends on the current annealing temperature controlled by the cooling scheme. At last, SA stops when it does not make significant progress over a number of iterations [2,3,12,13].

As described above, it is easy to see that simulated annealing is inherently sequential, thereby leading to long computation time, especially in the case of applying algorithm to problems with large search spaces. Naturally, a more efficient way to speed up the simulated annealing algorithm and make SA a more attractive solution for optimization problems is to add parallelism independent of the problem. However, different people may take different strategies in implementing a parallelizing SA, such as Mob parallel annealing [14], Time-homogeneous parallel annealing [15], Parallel recombinative simulated annealing (PRSA) [15] and Parallel systolic SA [16].

Our idea on parallelizing SA based on high-performance computing is to focus on two considerations. One good consideration is to abandon some search space and thus make SA always run in promising search space where a global optimization solution may exist. Another consideration is that SA's search space always remains unchanged. In this case, if different random number streams are used on each processor, then each processor will probably find an independent local minimum, and eventually we choose the smallest one among them as the best solution at the temperature of each cooling stage. Therefore, an important matter to think about is how to obtain different and good initial solutions on each participating processors and make each solution candidate independently and randomly move in its own search space with periodically communicating with each other.

Five strategies of parallelizing SA algorithms are illustrated in Fig. 1 and first implemented here on an IBM Beowulf PCs xSeries cluster using MPICH (a popular implementation version of MPI specification [17]). The principal feature of the MPI program is that a program consists of a (typically fixed) set of heavyweight processes (in general, each CPU can just only allow one process to run on it), each with a unique identifier (process ID from integers 0 to $P-1$, here supposed P is the total number of processes). If necessary, processes may interact by exchanging typed messages, by

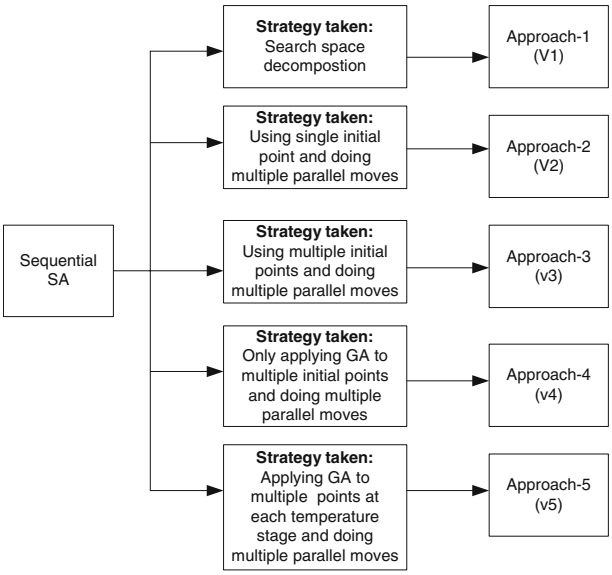

Fig. 1 Strategies used for parallelizing SA

engaging in collective communication operations, or by probing for pending messages. In case of our parallel computing platform, all processes are only executed on computing nodes of the IBM Beowulf PCs cluster. The number of computing nodes that may be used to run an MPI program depends upon the application submitted to the Portable Batch System (PBS) running on a head node.

2.2 Detailed descriptions of five approaches

Approach-1: Sequential SA algorithm works on the whole search space directly. If the search space is huge, it is not easy for SA to find the global optimum solution. Even though SA can occasionally succeed in getting one, it will take a very long time. It is a good strategy to decompose a large search space into smaller ones and then let SA work on them separately. This kind of Divide-and-conquer decomposition strategy has been applied in many search algorithms for both continuous and discrete problems [18–21]. Unpromising sub-spaces are excluded from further search, and promising ones are recursively decomposed and evaluated.

If $\vec{x}$ is one-dimension parameter x in objective function $f(\vec{x})$, then it is very easy to evenly partition the search space; If $\vec{x}$ is two-dimension parameters (x_1, x_2), then either of x_1 and x_2 can be used to equally partition search space. Suppose p is the number of desired search sub-space and x_2 is used to partition search space, therefore the search sub-space can be represented as follows:

Range of Sub-space 1: $x_1 \in (mn_1, mx_1)$ and $x_2 \in (mn_2, mn_2 + (mx_2 - mn_2)/p)$

Range of Sub-space 2:

$x_1 \in (mn_1, mx_1)$ and $x_2 \in (mn_2 + (mx_2 - mn_2)/p, mn_2 + 2 \times (mx_2 - mn_2)/p)$

Range of Sub-space 3:

$x_1 \in (mn_1, mx_1)$ and $x_2 \in (mn_2 + 2 \times (mx_2 - mn_2)/p, mn_2 + 3 \times (mx_2 - mn_2)/p)$

The rest may be deduced by analogy. Here mn_1, mx_1 represent the lower boundary and upper boundary of x_1, respectively; mn_2, mx_2 represent the lower boundary and upper boundary of x_2, respectively. If $\vec{x}$ is n-dimension parameters $(x_1, x_2, \ldots, x_n)$, then any one of parameters $x_1, x_2, \ldots, x_n$ may be chosen to evenly partition search space. The next search space is determined by parallel process 0. Parallel process 0 is responsible for collecting the minimum obtained by each parallel process and picks up the best of them as the promising solution, and then the range of the sub-space generated the best minimum solution is chosen as the next search space. Any one of parameters $x_1, x_2, \ldots, x_n$ in the next search space may be chosen to evenly partition the next search space.

Based on the idea mentioned above, Approach-1 may be described as follows:

1. An MPI program initially generates P processes that have ranks $0, 1, \ldots, P-1$.
2. Processes with rank 0 equally divide the search space into P parts and distribute the sub-space to P processes to let each process have a sub-space.
3. Each process runs a sequential SA algorithm and gets the optimum in its sub-space.[1]
4. The process with rank 0 is responsible for collecting the optimum obtained by each process and for deciding if the stop criterion is met or not.
   - 4.1 If the stop criterion[2] is met, the algorithm terminates.
   - 4.2 If not, process 0 has to determine the next search space and then goes back to step 2.

Each SA algorithm runs in its sub-space like a sequential SA. Different people may use different strategies to implement sequential SA. Suppose there already is an initial solution $\vec{x}(x_1, x_2, \ldots, x_n)$ to objective function $f(\vec{x})$. Our principle in implementation can be briefly described as follows:

```
current_solution ← initia_solution
current_cost t ← evaluate(current_solution)
T ← T_initial
while(stop_criterion_not_meet)
        for i = 1 to iterations(T)
           new_solution ← move(current_solution)
           Δcost t ← new_cost t − current_cost t
           if ( Δcost t ≤ 0 OR e^(−Δcost t/T) > Random(0,1) )
                 /* acceptnewsolution */
                 current_solution ← new_solution
                 current_cost t ← new_cost t
           endif
endfor
T ← newx_temp(T)
endwhile
```

The primary parameters used by each SA algorithm in each sub-space are as follows: Initial temperature: $T_{\text{initial}} = 1.0$;

[1] In terms of the used stop criterion, each parallel process may terminate in different ways, thereby making run time cost different. In order to guarantee that process 0 can collect the optimum minimum obtained by each parallel processes, synchronous point is set here after the 3rd step of V1 ends.

[2] Stopping criterion here is that the algorithm terminates when the difference between the previous and the latest optimal solution is less than or equal to 10^{-10}.

Temperature cooling rule *newx_temp*(T): $T_{t+1} = T_t \times 0.85$;
At each temperature, the number of iterations is equal to $1000 \times n$ (n is the number of parameters, $\vec{x}(x_1, x_2, \ldots, x_n)$);
Stop_criterion: if $((|f(\vec{x})_{G\min} - f(\vec{x})_{L\min}| < 1e-10) \quad || \ (T_{t+1} < 1e-04))$ is true, then SA terminates (Here $f(\vec{x})_{G\min}$ Represents the global minimum found so far; $f(\vec{x})_{L\min}$ Represents the global minimum newly found at current temperature);
In addition, function *move*(*current_solution*) is implemented as follows:

for($i = 0, i < n; i++$)
{
stepsize= ($mx_i - mn_i$)× *Random*(0, 1);
if(*Random*(0, 1) >= 0.5)
new_x$_i \leftarrow$ *current_x*$_i$+ *stepsize*;
else
new_x$_i \leftarrow$ *current_x*$_i$−*stepsize*;
if((*new_x*$_i$ < *mn*$_i$)|| (*new_x*$_i$ > *mx*$_i$))
Enforce_bound(*new_x*$_i$, *i*);
}

Above function *Enforce_bound*(*Parametervalue,i*) is implemented as follows:
{
if (*Parametervalue* >= *mx*$_i$) *Parametervalue* = *mx*$_i$ − *DBL_ EPSILON*;
else
if (*Parametervalue* > *mx*$_i$) *Parametervalue* = *mx*$_i$;
if (*Parametervalue* <= *mn*$_i$) *Parametervalue* = *mn*$_i$ + *DBL_ EPSILON*;
else
if (*Parametervalue* < *mn*$_i$) *Parametervalue* = *mn*$_i$;
}
(*Random*(0,1) is a random number generator between 0 and 1; mn_i, mx_i represent the lower boundary and upper boundary of x_i, respectively.)

Approach-2: A common approach to parallelizing simulated annealing is to generate several perturbations in the current solution simultaneously. Some of them, those with small variance, locally explore the region around the current point, while those with larger variances globally explore the feasible region. If each process has got different perturbation or move generation, each process will probably get a different solution at the end of iterations. Approach-2 may be described as follows:

1. The MPI program initially generates P processes that have ranks $0, 1, \ldots, P-1$.
2. The MPI program initially generates a starting point at random available for all processes and all processes set $T = T_0$.
3. Process with rank 0 randomly generates P different step sizes and distributes them to each participating process.
4. At the current temperature T, each process begins to execute iterative operations.
5. At the end of iterations, process with rank 0 is responsible for collecting the solution obtained by each process at current temperature and broadcasts the best solution of them among all participating processes.
6. If the termination condition is not met, each process cools the temperature and goes back to step 4, else algorithm terminates.

The core of the simulated annealing algorithm is the *Metropolis* procedure which simulates the annealing process at a given temperature T. The perturbation methods

to generate a new solution significantly affect the algorithm convergence. Suitable perturbation ways are expected to generate diverse solutions at each given temperature. In Approach-2, at the beginning the process 0 randomly generates different step sizes for each parameter with the following way and distributes them to each participating process:

$for(j = 0; j < P; j++)$
$\quad for(i = 0; i < n; i++)$
$\quad\quad \{$
$\quad\quad do$
$\quad\quad Stepsize[j][i] = (mx_i - mn_i) \times Random(0,1) + (mx_i - mn_i) \times (j/(2 \times P));$
$\quad\quad while(Stepsize[j][i] >= (mx_i - mn_i));$
$\quad\quad \}$

(mn_i, mx_i are the lower boundary and upper boundary of parameter x_i, respectively; n is the number of parameters of candidate solution $\vec{x}(x_1, x_2, \ldots, x_n)$ to objective function $f(\vec{x})$)

Therefore, at each parallel process each parameter has completely different step size from others parallel processes. Then the following perturbation ways are used in each parallel process:

$for(i = 0, i < n; i++)$
$\{$
$if(Random(0,1) >= 0.5)$
$new\_x_i \leftarrow current\_x_i + stepsize[i];$
$else$
$new\_x_i \leftarrow current\_x_i - stepsize[i];$
$if((new\_x_i < mn_i)||(new\_x_i > mx_i))$
$Enforce\_bound(new\_x_i, i);$
$\}$

However, sometimes the new solutions generated by perturbation functions did not make the condition expression $\left(\Delta \cos t \leq 0 \quad OR \quad e^{-\Delta \cos t/T} > Random(0,1)\right)$ be true and thus new neighborhood solution is not always accepted as next candidate solution. So the number of the accepted neighborhood solutions needs to be counted. In this case, the step sizes in perturbation ways should be changed according to the number of the accepted neighborhood solutions. Suppose that the number of iterations is set to $100 \times 10 \times n$ (this important control parameter is determined manually.), this means the step size can be changed at most 100 times and each parameters x_i has opportunity to generate 10 neighborhood perturbations. Therefore, the framework of *Metropolis* procedure can be described as follows:

$for(i = 0; i < 100; i++)$ // *Adjust ing stepsize*
$\quad for(j = 0; j < 10; j++)$ // *adjusting all parameters*
$\quad\quad for(k = 0; k < n; k++)$ // *adjusting one parameter*
$\quad\quad\quad \{$
$\quad\quad\quad codes\ to \overset{\cdots}{\underset{\cdots}{\ }} implement\ Metropolis\ procedure$
$\quad\quad\quad \}$

Suppose that for parameter x_i there are cc_i times to accept the neighborhood perturbation as next candidate solution. The step sizes for the parameter x_i will be changed in the following way:

$for(i = 0, i < n; i++)$
{
$rate = cc_i/10;$
$if(rate > 0.6) stepsize[i] = stepsize[i] \times (1 + 5 \times (rate - 0.6));$
$if(rate < 0.4) stepsize[i] = stepsize[i]/(1 + 5 \times (0.4 - rate));$
}

Of course, in general, the above step size $stepsize[i]$ for each parameter x_i should always be within the range of $(mx_i - mn_i)$. This work can be done by an additional function. Therefore, some parallel processes with small variances in step sizes, will locally explore the region around the current point, while those parallel processes with larger variances in step sizes will globally explore the feasible region.

Approach-3: The quality of an SA's performance is usually affected by the configuration of the starting solution. We may further improve Approach 2 by changing a single start-solution into multiple start-solutions. Thus, we get the Approach 3 and may think of it as a multistart search technique. To get Approach 3, we just need to modify the step 2 in Approach 2 as follows:

2. Process with rank 0 initially generates P different starting points at random and distributes them to each participating process and all processes set $T = T_0$.

In process 0, multiple different starting points are randomly generated as follows:

$for(i = 0; i < P; i++)$
  $for(j = 0; j < n; j++)$
  $x[i][j] = mn_j + (mx_j - mn_j) \times Random(0, 1);$

(mn_j, mx_j are the lower boundary and upper boundary of parameter x_j, respectively; n is the number of parameters of candidate solution $\vec{x}(x_1, x_2, \ldots, x_n)$ to objective function $f(\vec{x})$)

Approach-4 and Approach-5: Genetic algorithms are population-based optimization algorithms based on Darwinian models of natural selection and evolution. The basic idea behind GA is that an initial population of candidate solutions of size N is chosen at random and each solution is evaluated according to the specified optimization function [21]. If genetic operators (Crossover and Mutation) are chosen properly and applied to a population by a specific number of iterations (generations), then eventually the population will have better solutions. Thus, GA improves the whole population. In Approach 3, the initial multiple starting-solutions are randomly generated to reduce the effect caused by the configuration of the starting solution. In fact, those starting solutions may not be very good and can be further optimized. In order to guarantee the quality of initial starting solutions, GA may be applied to those initial solutions. Therefore, Approachs 4 and 5 may be elaborated as follows, respectively:

Apprach-4:

1. The MPI program initially generates P processes that have ranks $0, 1, \ldots, P - 1$.
2. Process with rank 0 initially generates P random start-solutions and runs GA by a fixed number of iterations (suppose that the maximum generation is set to 1,000 or more). After the GA ends, process with rank 0 distributes P individuals in the final population to each participating process as an initial starting solution.
3. All processes set current temperature $T = T_0$;
4. At the current temperature T, each process begins to execute those iterative operations.

5. At the end of the iterations, process with rank 0 is responsible for collecting the solution obtained by each process at current temperature and broadcasts the best solution of them among all participating processes.
6. If the termination condition is not met, each process cools the temperature and goes back to step 4, otherwise the algorithm terminates.

Approach-5:

1. The MPI program initially generates P processes that have ranks $0, 1, \ldots, P-1$.
2. Process with rank 0 initially generates P random start-solutions and runs GA by a fixed number of iterations (suppose that the maximum generation is set to 1,000 or more). After the GA ends, process with rank 0 distributes P individuals in the final population to each participating process as an initial starting solution.
3. All processes set current temperature $T = T_0$;
4. At the current temperature T, each process begins to execute those iterative operations.
5. At the end of the iterations, process with rank 0 is responsible for collecting the solutions obtained by each process at current temperature and then employing a GA to evolve them by a fixed number of generations and finally broadcasting the best solution of the population achieved by GA among all participating processes.
6. If the termination condition is not met, each process cools the temperature and goes back to step 4, otherwise the algorithm terminates.

Inside above Approachs 4 and 5, in the case of minimizing $f(\vec{x})$ is subject only to lower- and upper-bound constraints on variables, the designed GA has used real parameters as chromosome codes and may work as follows:

1. Generate P individuals at random as initial solutions.
2. Evaluate the initial population.
3. Apply crossover operators to the current population to get $2P$ individuals (P parent individuals plus P children individuals).
4. Apply mutation operators to each child individual and evaluate the P mutated individuals.
5. According to the fitness value, by tournament competition strategy, select P individuals from $2P$ individuals as the next evolved population.
6. Determine if the iteration reaches the maximum generation or not:

   6.1. If yes, the algorithm terminates.
   6.2. If no, the algorithm needs to do following operation:

   - If the best stored individual is not updated consecutively and the number of counters cumulatively reaches 10 times (or 30 times or 50 times), 10% (or 30% or 50%) individuals located at the bottom of the next evolved population will be replaced by randomly generated individuals, and then the algorithm goes back to step 3 and continues.[3]

[3] In our algorithm, the best individual found so far is stored additionally. At the each generation, if the best individual currently generated in population is better than the previously stored individual in evaluation fitness, then it will replace the stored best individual. A counter is set to count times. Suppose the best stored individual has not been consecutively updated by 10 times, then 10% individuals located at the bottom of the population will be replaced by randomly generated individuals. The rest (30% or 50%) may be deduced by analogy. Obviously this operation can be done after the individuals in population are sorted in terms of evaluation fitness.

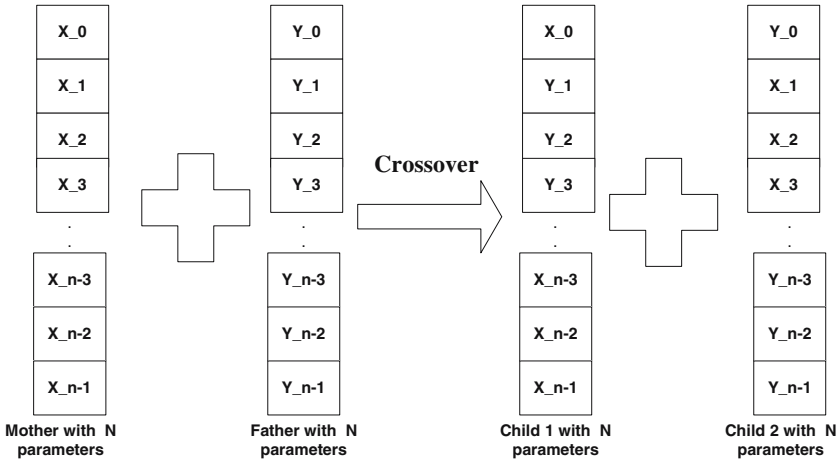

Fig. 2 Crossover operation with *Num_crossover* = 3 and *First_position* = 2

If GA is applied to a problem, in general, there are at least four issues, namely, chromosome code, fitness evaluation, selection strategy and genetic operations, which need to be addressed. As mentioned above, SA is an individual-based optimization algorithm and algorithm only optimize one candidate solution to optimization function $f(\vec{x})$, so the candidate solution $\vec{x}(x_1, x_2, \ldots, x_n)$ can be coded as a vector(one dimensional array). In GA, suppose that an initial population comprise p individuals, so the individuals will have to be coded with two-dimensional array, such as $individual[i][j], (0 <= i < p; 0 <= j < n)$ (where p is population size and n is the number of parameters). Therefore, the fitness evaluation function is naturally equivalent to the optimization function $f(\vec{x})$ itself. Genetic operations generally include crossover and mutation. Before crossover operation begins, it is necessary to know where the crossover position will be and how many parameters will be used to cross over. In the implementation of GA, the algorithm always randomly generates a number *Num_crossover* that is no more than half of a chromosome length. In addition, the algorithm also randomly chooses the first position in terms of the following rule:

$$Num\_crossover = floor\left(Num\_parameter/2 * Random\_num[0,1]\right)$$

$$First\_position = floor(1 + Random\_num[0,1] * (Num\_parameter - Num\_crossover))$$

(* *floor*(*double x*) *is one of C/C++ mathematical library functions and* computes the largest integral value not greater than x.)

For example, Fig. 2 shows the process of crossover operation with *Num_crossover* = 3 and *First_position* = 2 (supposing here the number of parameters is $N \geq 7$ at lease.). Mutation operation may be done as follows: mutation operator at first needs to randomly choose a parameter $x_i, (1 < i < n)$ from a child individual $\vec{x}(x_1, x_2, \ldots, x_n)$ and replace it with a randomly generated number. Of course, the boundary condition $mn_i <= x_i <= mx_i$ should be always satisfied. After the population is operated by crossover and mutation, the number of parent population plus offspring is definitely more than the current parent population size. Tournament competition strategy is used to keep the size of next parent population unchanged.

Approach-5 is based on Approach-4 and just only employs a different way in STEP 5. Obviously, in the case of Approach-5, GA will be used to further optimize the solutions collected from each process at each cooling temperature stage and always pick up the best solution as optimization target of next temperature step.

3 Experimental results and performance analysis

3.1 A sample test problem

In practice, many optimization problems deal with a large number of variables (up to tens or hundreds or more) and/or a very large number of local minima. In these situations, traditional optimization methods implemented in software running on a single computer generally offer low efficiency and limited reliability. Thus, in order to better test our proposed algorithms, for instance, we selected the following significant N-Dimensional general test function:

$$f(\bar{x}) = (1/2)\sum_{j=1}^{N}\left(x_j^4 - 16x_j^2 + 5x_j\right), \tag{1}$$

as an objective function, where N-dimensional parameters may range from $-\infty$ to $+\infty$. For this test function, the number of local minima is 2^N, but the number of global minima is just 1. Cetin et al. [23], by using Terminal Repeller Unconstrained Subenergy Tunneling (TRUST), which is a kind of deterministic method, have found the function's global minimum that is located at $\left[\overline{x_{GM}}\right] = [-2.90354, -2.90354, \ldots, -2.90354]$. Because each run of a stochastic optimization algorithm may find a different optimal/good solution, we really do not know whether or not the obtained solution is close to the true global optimal solution. The determination criterion for the attainable solution quality used by us in all experimental tests may be described as follows:

If the error between each parameter and –2.90354 is less than 0.001, then we may think that the optimization algorithm has succeeded to find the global optimum (*the number of success is added 1*); *otherwise, the optimization algorithms failed to find the global optimum.*

For example, one satisfactory global optimum solution to the test function in case of $N = 50$ found by our approach-V5 is given in Table 1. We think approach-V5 has succeeded to find the global optimum solution to the test problem in terms of above determination criterion. One of our motivations is to verify whether the five different parallel SA approaches can effectively reach the global minimum of the above test function. More importantly, we need to know which is the best approach to some function optimization problems in terms of success rate and convergence speed as the search space increasingly extends. The following tests have been done on the IBM Beowulf PCs Linux cluster. This kind of high-performance computer consists of 2 head nodes, 2 manager nodes and 96 computing nodes, each node with 1GB memory has two Pentium III 1 GHZ processors. Networking in the cluster is accomplished with Myrinet with full-duplex 2 + 2 Gigabit/second links.[4] In all experiments, it is

[4] On Beowulf PCs cluster the parallel software developed using MPI can only allow one parallel process to run on one CPU. This is a restriction of MPI parallel programs. A PBS job management system (a kind of job scheduler) is installed on our IBM Beowulf PCs Cluster. Under this circumstance, if

Table 1 Sample solution to the used test function found by Approach-V5

Function value *50 parameter values*	−1958.30828518835	
−2.9035346316028	−2.90353370333875	−2.90353415109338
−2.90353437349436	−2.90353466888656	−2.9035332962738
−2.9035355313778	−2.90353397771527	−2.90353377506699
−2.90353407783128	−2.90353419327455	−2.90353429236886
−2.90353364675147	−2.90353422397274	−2.90353379298958
−2.90353434335605	−2.90353492607259	−2.90353393083044
−2.90353429102594	−2.90353356769459	−2.90353406882167
−2.9035341619688	−2.90353441755032	−2.90353420362525
−2.90353377519349	−2.90353406322817	−2.90353390773739
−2.90353412449576	−2.90353405020605	−2.90353373204054
−2.90353287974997	−2.90353389509765	−2.90353512917096
−2.90353413618296	−2.90353484496602	−2.90353437264664
−2.90353315775705	−2.90353458733501	−2.90353316152059
−2.90353382960012	−2.90353394186907	−2.90353354737074
−2.90353406912158	−2.90353375294677	−2.90353383431399
−2.9035341329692	−2.90353404615635	−2.90353465063491
−2.90353409266615	−2.90353490429537	

worth pointing out that the termination criteria used for all approaches proposed in this paper are given as follows:
1. For V1 approach, the determination criterion for sequential SA inside V1 is as follows:
Stop_criterion: if $((|f(\vec{x})_{G\min} - f(\vec{x})_{L\min}| < 1e-10)$ || $(T_{t+1} < 1e-04))$ *is true, then SA terminates (Here* $f(\vec{x})_{G\min}$ *Represents the global minimum found so far;* $f(\vec{x})_{L\min}$ *Represents the global minimum newly found at current temperature);*
But for V1, V2, V3, V4 and V5 approach, their common stop criterion is as follows:
Algorithm terminates when the difference between the previous and the latest global optimal solution is less than or equal to 10^{-10}.

Inside V4 and V5 approach, the termination criterion for GA is the maximum generation. The value of maximum generation is set according to the complexity of objective functions.

3.1.1 Increasing search space dimensions

For the above test function (1), if the variable range of parameters is not too big, increasing the search space dimensions may be a good way to test these five different methods' performance. When the parameter interval falls into the range $[-10^4, 10^4]$, experiments with three cases have been fully done with the number of dimension $N = 10, N = 20$ and $N = 30$, respectively. Experimental results are shown in Tables 2–4, respectively (For example, in the tables, success rate 3/10 indicates this approach has been used successfully to find the global minimum 3 times out of 10

Footnote 4 continued
a MPI parallel program is running on multiple CPUs, PBS system will not allow other users' jobs to launch/run on those CPUs occupied by the running MPI program. Therefore both the program wall clock time and CPU time cost can be measured without any interference from other users' jobs. After the MPI parallel program ends, PBS will schedule the jobs in queue according to some rules, such as "first come first serve".

Table 2 In the case of $N = 10$, success rates of 5 versions running on Beowulf Cluster

	Number of processes						
	10	20	40	80	120	160	192
V1	5/10	2/10	6/10	3/10	4/10	5/10	6/10
V2	3/10	6/10	6/10	3/10	2/10	3/10	2/10
V3	4/10	2/10	2/10	3/10	2/10	2/10	2/10
V4	4/10	4/10	2/10	2/10	1/10	1/10	1/10
V5	10/10	10/10	10/10	10/10	10/10	10/10	10/10

Table 3 In the case of $N = 20$, success rates of 5 versions running on Beowulf Cluster

	Number of processes						
	10	20	40	80	120	160	192
V1	6/10	3/10	4/10	7/10	5/10	5/10	3/10
V2	2/10	3/10	2/10	4/10	1/10	1/10	1/10
V3	3/10	1/10	3/10	1/10	2/10	2/10	1/10
V4	3/10	2/10	2/10	0/10	0/10	0/10	1/10
V5	10/10	10/10	10/10	10/10	10/10	10/10	10/10

Table 4 In the case of $N = 30$, success rates of 5 versions running on Beowulf Cluster

	Number of processes						
	10	20	40	80	120	160	192
V1	3/10	5/10	4/10	8/10	6/10	8/10	8/10
V2	1/10	2/10	1/10	2/10	1/10	1/10	1/10
V3	4/10	2/10	3/10	0/10	2/10	0/10	0/10
V4	1/10	1/10	0/10	0/10	0/10	0/10	0/10
V5	10/10	10/10	10/10	10/10	10/10	10/10	10/10

runs.). It is easily observed from Tables 2–4 that each one of the five approaches can successfully find the global minimum. However, different approaches have yielded different success rates in reaching equilibrium. Only in terms of success rate, it is easy to see the fact that the fifth approach, V5, is the most efficient and robust, as it never failed to find the global optimal solution in any of the runs, while some of the other methods did fail frequently. Additionally, the first approach, V1, seems to be a good method because it works well in most cases. However, we found out that its run time cost is significantly increased in all cases when the number of processes is equal to 10,20,40,80,120,160,192, respectively as the search space dimensions grow. Figure 3 shows us an example in the case of the number of CPUs equals to 10, 160 and 192, respectively. More badly, this indicates that V1 will lead to costs over a long period of time if the random search space grows fast.

For better understanding of the performance of V5, when we kept increasing search space dimensions up to $N = 50$ and $N = 70$, we found, to our surprise, that V5 never failed to converge to the global optimum solution. Moreover, it is worthy to note that the run time cost of v5 almost linearly increases as the number of processes grows in the case of small search space dimensions. We may see this point from Fig. 4, but the run time cost of V5 parabolically changes as the number of processes grows in the case of bigger search space dimensions. This interesting result may be observed in Fig. 5 and 6 as well.

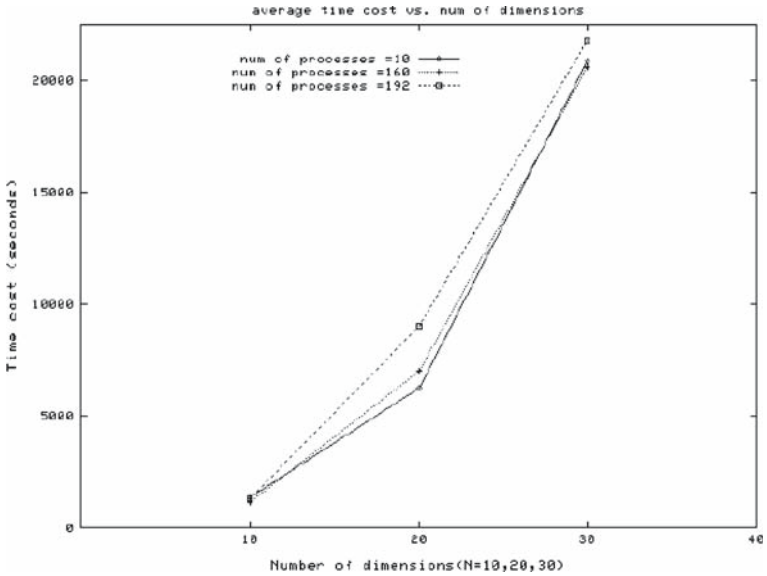

Fig. 3 Average run time versus number of dimensions for V1, in case of the number of processes equal to 10,160,192 (10 runs each case)

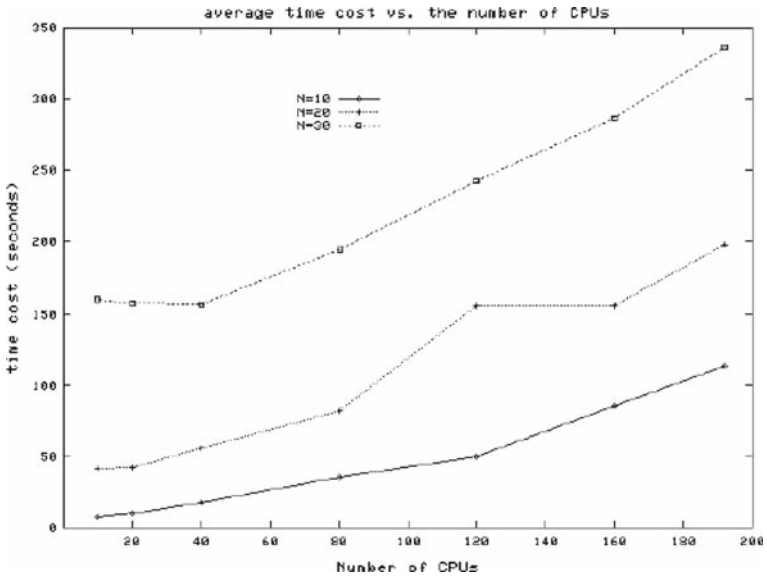

Fig. 4 In the case of $N = 10, 20, 30$, V5's average run time of 10 runs versus the number of processes

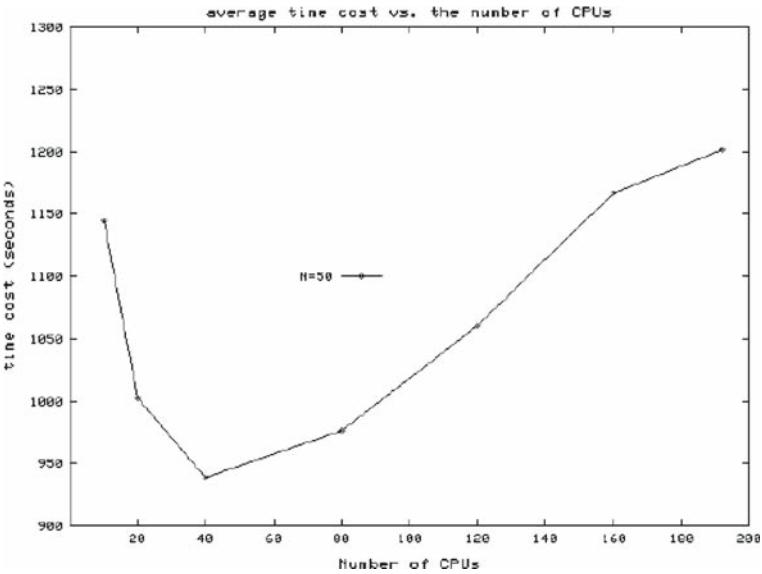

Fig. 5 In case of $N = 70$, V5's average run time of 10 runs versus the number of processes

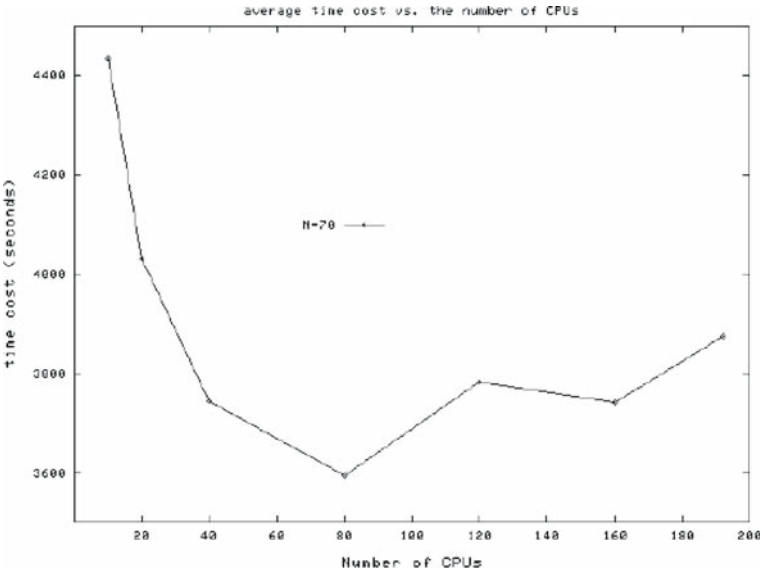

Fig. 6 In case of $N = 70$, V5's average run time of 10 runs versus the number of processes

3.1.2 Increasing parameter interval

For the above test function (1), another alternative to augment search space is to increase parameter interval. When we selected $N = 50$ and considerably varied the parameter interval from $[-10^4, 10^4]$ to $[-10^{10}, 10^{10}]$, we found out that the other four approaches seemed to be ineffective in most cases and generally failed to converge to

Table 5 In the case of $N = 50$, success rates of V5 running on Beowulf Cluster

	Number of processes							
	10	20	40	64	80	120	160	192
R1	10/10	10/10	10/10	10/10	10/10	10/10	10/10	10/10
R2	10/10	10/10	10/10	10/10	10/10	10/10	10/10	10/10
R3	10/10	10/10	10/10	10/10	10/10	10/10	10/10	10/10
R4	10/10	10/10	10/10	10/10	10/10	10/10	10/10	10/10
R5	10/10	10/10	10/10	10/10	10/10	10/10	10/10	10/10
R6	10/10	10/10	10/10	10/10	10/10	10/10	10/10	10/10
R7	10/10	10/10	10/10	10/10	10/10	10/10	10/10	10/10

(R1, R2, R3, R4, R5, R6 and R7 represent the parameters range $[-10^4, 10^4]$, $[-10^5, 10^5]$, $[-10^6, 10^6]$, $[-10^7, 10^7]$, $[-10^8, 10^8]$, $[-10^9, 10^9]$ and $[-10^{10}, 10^{10}]$, respectively.)

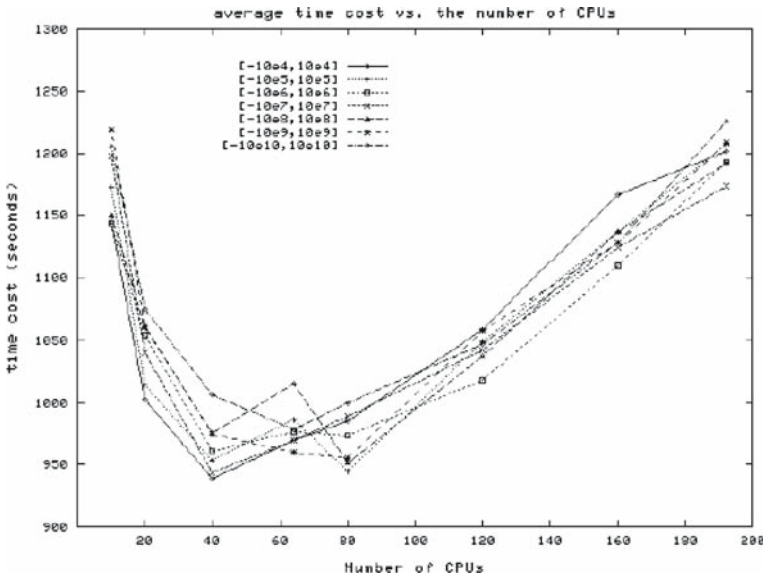

Fig. 7 In case of different parameter range and $N = 50$, V5 run time cost versus the number of processes used

the known global optimum solution. Only V5 could successfully find the global minimum located at $\lceil \overline{x_{GM}} \rceil = \lceil -2.90354, -2.90354, \ldots, -2.90354 \rceil$ and its success rate is very high with 100% rate. Experimental results are presented in Table 5. Additionally, we also found that the average run time cost out of 10 runs of V5 parabolically varied with the number of CPUs used as the parameter interval increased (this means the search space expands considerably). Experimental results are given in Fig. 7. This shows that there must be an optimal number of processors p that minimizes the program execution time. This optimal number of processors p would be appearing between 40 and 100 according to the information released in Fig. 7. This simultaneously implies that it is not possible to further reduce the run time cost of V5 by gradually increasing the number of CPUs used. We will discuss this point in detail in Sect. 3.4.

Table 6 The success rates of V1 running on Beowulf Cluster

	Number of processes						
	10	20	40	80	120	160	192
F1	5/10	4/10	4/10	8/10	7/10	6/10	7/10
F2	3/10	7/10	4/10	7/10	7/10	4/10	7/10
F3	3/10	7/10	3/10	3/10	10/10	7/10	8/10
F4	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F5	6/10	9/10	9/10	10/10	5/10	6/10	10/10
F6	2/10	3/10	0/10	3/10	1/10	3/10	7/10
F7	0/10	1/10	1/10	3/10	2/10	2/10	3/10
F8	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F9	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F10	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F11	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F12	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F13	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F14	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F15	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F16	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F17	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F18	10/10	10/10	10/10	10/10	10/10	10/10	10/10

3.2 More tests for checking the generality of the five approaches

The efficiency of algorithms developed in this paper needs to be tested on various functions, with different orders of difficulty. Each test function may yield different number of parameters and each parameter may yield different ranges. To guarantee the generality of the five approaches, initial points will have to be generated randomly and must fall into the range of the parameters. Also, the computer's machine time is used as a random seed in the implementation of all approaches. Thus characteristic of randomness is absolutely guaranteed for the five approaches.

For purposes of comparison, the behavior of each approach is determined by an extensive computational experiment by using some well-known global optimization test functions. The mathematical representations of these functions are presented in Appendix A and the experimental results are given in Tables 6–10, respectively. These functions are well-known test problems in the literature and are commonly used by other researchers to test their search algorithms [24,25]. Most of these test functions have low dimensionality, but they have several local and global optima and large flat regions enclosing the global optima. These characteristics generally make it difficult to find the global optima of those test functions.

From Tables 6–10, it can be seen that V5 performed significantly better than other methods in success rate and in most cases reached the global optimum solution in all runs for 18 cases of test problems. Especially, in the case of function F7 which is of a high degree of difficulty, other algorithms including V1 did not work well. Though for V5, there is just a low success rate if a few CPUs used. However, if the number of CPUs used increases, the success rate for V5 improves. To really test the efficacy of V5, it must be tried on extremely difficult problems. This is because almost any algorithm will probably solve an easy problem, but the real advantages of one algorithm over another will become evident in when solving those very difficult problems. Thus, we continued to increase the complexity of F7 by using bigger N-dimensional

Table 7 The success rates of V2 running on Beowulf Cluster

	Number of processes						
	10	20	40	80	120	160	192
F1	0/10	0/10	0/10	0/10	0/10	0/10	0/10
F2	0/10	0/10	0/10	0/10	0/10	0/10	0/10
F3	0/10	0/10	0/10	0/10	0/10	0/10	0/10
F4	2/10	0/10	0/10	0/10	0/10	0/10	0/10
F5	0/10	0/10	0/10	0/10	0/10	0/10	0/10
F6	0/10	0/10	0/10	0/10	0/10	0/10	0/10
F7	0/10	0/10	0/10	0/10	0/10	0/10	0/10
F8	5/10	7/10	8/10	6/10	7/10	7/10	7/10
F9	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F10	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F11	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F12	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F13	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F14	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F15	9/10	10/10	10/10	10/10	10/10	10/10	10/10
F16	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F17	9/10	10/10	10/10	10/10	10/10	10/10	10/10
F18	10/10	10/10	10/10	10/10	10/10	10/10	10/10

Table 8 The success rates of V3 running on Beowulf Cluster

	Number of processes						
	10	20	40	80	120	160	192
F1	0/10	0/10	0/10	0/10	0/10	0/10	0/10
F2	0/10	0/10	0/10	0/10	0/10	0/10	0/10
F3	0/10	0/10	0/10	0/10	0/10	0/10	0/10
F4	2/10	1/10	1/10	0/10	0/10	0/10	0/10
F5	0/10	0/10	0/10	0/10	0/10	0/10	0/10
F6	0/10	0/10	0/10	0/10	0/10	0/10	0/10
F7	0/10	0/10	0/10	0/10	0/10	0/10	0/10
F8	7/10	10/10	10/10	10/10	10/10	10/10	10/10
F9	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F10	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F11	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F12	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F13	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F14	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F15	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F16	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F17	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F18	10/10	10/10	10/10	10/10	10/10	10/10	10/10

parameter-spaces of $N = 10, 12, 14$, or 16 and got the experimental results listed in Table 11. Experimental results show that V5 can find the global optimal/near-optimal solution to the test problem F7 of high difficulty and also indicate that V5 can have potential ability to solve some extremely difficult function problems.

Table 9 The success rates of V4 running on Beowulf Cluster

	Number of processes						
	10	20	40	80	120	160	192
F1	7/10	8/10	6/10	7/10	7/10	5/10	6/10
F2	3/10	4/10	0/10	1/10	1/10	4/10	2/10
F3	2/10	0/10	1/10	0/10	0/10	1/10	0/10
F4	9/10	10/10	10/10	10/10	10/10	10/10	10/10
F5	2/10	6/10	0/10	4/10	3/10	4/10	6/10
F6	0/10	0/10	0/10	1/10	0/10	3/10	0/10
F7	0/10	0/10	0/10	0/10	0/10	0/10	0/10
F8	10/10	10/10	10/10	10/10	9/10	10/10	10/10
F9	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F10	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F11	10/10	10/10	9/10	10/10	10/10	10/10	10/10
F12	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F13	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F14	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F15	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F16	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F17	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F18	10/10	10/10	10/10	10/10	10/10	10/10	10/10

Table 10 The success rates of V5 running on Beowulf Cluster

	Number of processes						
	10	20	40	80	120	160	192
F1	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F2	10/10	9/10	9/10	10/10	10/10	10/10	10/10
F3	10/10	9/10	8/10	10/10	10/10	10/10	9/10
F4	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F5	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F6	7/10	7/10	10/10	10/10	10/10	10/10	10/10
F7	0/10	0/10	1/10	2/10	8/10	8/10	10/10
F8	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F9	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F10	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F11	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F12	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F13	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F14	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F15	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F16	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F17	10/10	10/10	10/10	10/10	10/10	10/10	10/10
F18	10/10	10/10	10/10	10/10	10/10	10/10	10/10

3.3 Comparisons with other global optimization methods

In general, it is difficult to find sections in published results in some literatures to compare with our results. This is due to the fact that different people used different test functions in their work and algorithms were implemented on different computing platforms with different aims. For example, Esin and Linet [11] implemented parallel

Table 11 Success rates of V5 in case of high degree of difficulty

	Number of processes							Best minimum
	10	20	40	80	120	160	192	
N = 10	0/10	1/10	1/10	1/10	3/10	5/10	7/10	0
N = 12	0/10	0/10	0/10	0/10	0/10	1/10	1/10	0
N = 14	0/10	0/10	0/10	0/10	0/10	0/10	0/10	0.41482
N = 16	0/10	0/10	0/10	0/10	0/10	0/10	0/10	0.89891

SA procedures and evaluated them on large-scale test functions with the aim of locating the global optima within efficient computation times. Ellen et al. [7] implemented a parallel SA on a hypercube multiprocessor using speculative computation to obtain speedup without sacrificing the general applicability of simulated annealing or the high quality solutions, and the parallel algorithm was applied to another important parallel processing problem of task assignment. Cem [26] implemented a hybrid parallel SA algorithm to optimize store performance. Almost all parallel SA algorithms in the literature have concentrated on specific problems. However, finally we discovered some work in some literature that was done with the same set of test functions and similar destinations [24,25,27].

3.3.1 The number of function evaluations needed for each method

In general, the number of function evaluations needed to reach the minimum can be used to test the efficiency of global optimization methods because it is machine-independent. For comparison purposes, the results in Table 12 generated by the Adaptive Random Search, Simulated Annealing algorithm and Genetic Algorithm are taken from [25] and [27]. Rosenbrock classical test functions in 2 and 4 dimensions used in those publications are as follows:

$$f(x_1,x_2) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2, \tag{2}$$

$$f_(x_1,x_2,x_3,x_4) = \sum_{k=1}^{3} 100(x_{k+1} - x_k^2)^2 + (1 - x_k)^2, \tag{3}$$

The admissible domains of the functions were $X_i \in [\pm 2000,\ \pm 2000] \forall i$ for 2 dimensions and $X_i \in [\pm 200,\ \pm 200] \forall i$ for 4 dimensions. It is easily observed from Table 12 that a group of fixed points as starting points is used in ARS, SA and GA. For V5 unlike other methods used in comparison, each test began with a randomly generated initial solution. The results out of 100 trials of v5 are presented in Table 13.

Apparently, in most cases, minima generated by V5 are much better than those generated by other methods. However, it is worth noting that the number of function evaluations performed by V5 is always big. Perhaps the reason is that different stopping criteria may be used among the methods listed in Table 14. In V5, two function evaluations were done by both SA and GA at each temperature cooling stage. As mentioned above, the stopping criteria used by SA inside V5 is that the algorithm terminates when the difference between the previous and the latest best solution is less than or equal to 10^{-10}. The termination condition used by GA inside V5 is the

Table 12 Comparison of the ARS, SA and GA for Rosenbrock Valley functions[a]

Starting point	Method					
	Function evaluations			Final function value		
	ARS	SA	GA	ARS	SA	GA
2 dimensions						
1001,1001	3411	500001	2389	1586.4	1.8E−10	1.2E−12
1001,−999	131841	508001	2214	8.6E−9	2.6E−9	2.3E−10
−999,−999	15141	524001	3254	1.2E−8	1.2E−9	4.4E−11
−999,1001	3802	484001	3412	583.2	4.2E−8	3.4E−10
1443,1	181280	492001	2115	4.7E−10	1.5E−8	3.4E−10
1,1443	2629	512001	5781	1468.9	1.6E−9	1.2E−10
1.2,1	6630	488001	1548	5.5E−7	2.0E−8	2.2E−18
4 dimensions						
101,101,101,101	519632	1288001	228534	1.9E−6	5.0E−7	4.8E−9
101,101,101,−99	194720	1328001	213422	1.7E−6	1.8E−7	2.1E−9
101,101,−99,−99	183608	1264001	264521	3.8E−6	5.9E−7	2.2E−8
101,−99,−99,−99	195902	1296001	299321	2.3E−6	7.4E−8	3.0E−9
−99,−99,−99,−99	190737	1304001	44567	2.7E−6	3.3E−7	5.7E−9
−99,101,−99,101	4172290	1280001	234312	2.6E−6	2.8E−7	3.9E−8
101,−99,101,−99	53878	1272001	193134	3.7	2.3E−7	4.1E−8
201,0,0,0	209415	1288001	182131	1.1E−6	7.5E−7	3.0E−8
1,201,1,1	215116	1304001	283946	1.2E−6	4.6E−7	5.8E−8
1,1,1,201	29069006	1272001	214312	2.2E−6	5.2E−7	4.3E−8

[a] Results taken from Corana et al. [25] and Hussain and Al-Sultan [27]

Table 13 Results of V5 for Rosenbrock Valley functions

Number of dimensions		Num of processes (10 runs each case)				
		10	20	40	60	80
2	Function evaluations[a]	262,440	398,721	1,133,796	1,826,782	1,763,232
	Best minimum[a]	3.1E−9	1.3E−9	2.3E−10	1.6E−9	1.3E−10
4	Function evaluations[a]	4,176,801	5,804,293	11,608,858	14,177,531	21,256,251
	Best minimum[a]	3.2E−10	3.2E−10	3.0E−9	1.0E−10	1.1E−11
		100	120	140	160	192
2	Function evaluations[a]	2,834,414	1,635,787	3,968,373	3,526,375	7,382,010
	Best minimum[a]	6.9E−11	9.9E−09	2.1E−11	1.1E−09	3.7E−11
4	Function evaluations[a]	21,178,909	1,8944,175	34,454,521	30,747,552	30,312,020
	Best minimum[a]	3.9E−11	4.6E−11	7.7E−11	1.6E−11	1.8E−10

[a] The best minimum generated by V5 of 10 runs and the number of function evaluations accordingly given by V5

maximum generation that may be manually adjusted depending on the complexity of the test functions. It is not a good idea to use a fixed maximum generation on all of the test functions. Thus, we believe that, in some cases, a decision cannot be made to determine which method is the best one only in terms of the number of function evaluations. In fact, some other factors, such as the accuracy of the obtained global optimal solution and the success rate to obtain satisfactory global optimum solution, need to be considered simultaneously as well.

Table 14 Methods used in comparison

Method	Name	Reference
ARS	Adaptive random search	Masri et al. [24]
SA	Simulated annealing	Corana et al. [25]
GA	Genetic algorithm for multimodal function	Hussain and Al-sultan [27]
V5	Highly hybrid genetic algorithm and simulated annealing	Dingjun and Park (2006) (this paper)

Table 15 Methods used in comparison

Method	Name	Reference
HSGT	Hybrid scatter genetic tabu search	Trafalis and Kasap [24]
HSG	Hybrid scatter genetic search	Trafalis and Al-Harkan [25]
SA	Simulated annealing	Goffe et al. [27]
GENOCOP	Genetic algorithm for numerical optimization with constraints	Michalewicz and Janikow [25]
V5	Highly hybrid genetic algorithm and simulated annealling	Dingjun et al. (2006) (this paper)

3.3.2 Success rate and optimum solution quality

The success rate may be used as a measure to verify the robustness of an algorithm. In general, we cannot rely on the CPU time cost which is machine-dependent, as our measure of the efficiency of algorithms. However, if there is no special limit on algorithm run time costs, we may consider making some comparisons in solution quality with other global optimization algorithms so as to ascertain some advantages or disadvantages of algorithms. Thus, we have also compared the performance of V5, which is the best approach we proposed here, with some other recent algorithms in the literature that are shown in Table 15 in terms of their success rate, CPU time cost and solution quality. Some of the test functions listed in Appendix A have been tested using other global optimization methods. Comparable results are presented in Tables 16 and 17. It is easily observed that V5 is much better than other methods used in comparisons just in terms of the success rate and the optimum solution quality.

3.4 Further performance analyses

According to the analyses above, V5 may be viewed as the best algorithm in global function optimization problems. In this section, we will further analyze its inherent character and measure its performance in parallelism. Many metrics are used for measuring the performance of a parallel program running on a high-performance computer [28]. The most commonly used measurements are elapsed time, price/performance, speed-up[5] and efficiency.[6]

[5] In this paper, we used the relative speed-up. It may be defined as follows: $S_P^{\text{relative}} = T_2/T_P$, T_2 is the execution time of a parallel program on a high-performance computer with 2 CPUs used; T_P is the execution time of a parallel program on a high-performance computer with P CPUs used.

[6] In this paper, we used the relative efficiency. It may be defined as follows: $\eta_P^{\text{relative}} = T_2 \times 2/T_P \times P = S_P^{\text{relative}} \times 2/P$. Efficiency close to unity means that you are using your hardware effectively; low efficiency means that you are wasting resources.

Table 16 The success rate for each test problem and method

Problem	HSGT[a]	HSG[a]	SA[a]	GENOCOP[a]	V5[b]
F8	100/100	100/100	100/100	100/100	100/100
F9	100/100	100/100	100/100	100/100	100/100
F10	61/100	5/100	58/100	100/100	100/100
F11	100/100	100/100	100/100	100/100	100/100
F12	70/100	99/100	100/100	100/100	100/100
F13	100/100	100/100	100/100	100/100	100/100
F14	100/100	100/100	20/100	100/100	100/100
F15	100/100	98/100	5/100	100/100	100/100
F16	100/100	45/100	100/100	93/100	100/100
F17	100/100	100/100	100/100	100/100	100/100
F18	100/100	84/100	100/100	100/100	100/100
F19	100/100	100/100	100/100	100/100	100/100
F20	100/100	52/100	74/100	100/100	100/100
F21	76/100	19/100	7/100	100/100	100/100
F22	36/100	4/100	0/100	74/100	100/100

[a] Results taken from Trafalis and Kasap [24]

[b] V5 was executed 10 times, in the case of the number of processes equal to 10,20,40,60,80, 100,120,140,160,192, respectively. Thus the results of 100 trials are given

Table 17 CPU time and percentage deviation from global optimum

Problem	CPU time cost (s)					Percentage deviation from global optimum			
	HSGT[a]	HSG[a]	SA[a]	V5[b]		HSGT[a]	HSG[a]	SA[a]	V5
				Mini	Maxi				
F8	2.39	1.30	5.89	3.81	292.14	0.00	0.02	0.45	0.00
F9	1.90	1.01	4.53	3.01	281.75	0.00	0.05	0.30	0.00
F10	4.45	1.17	5,33	3.62	428.56	0.30	35.95	0.07	0.00
F11	1.62	0.88	3.91	2.64	184.86	0.00	0.01	0.03	0.00
F12	5.67	1.44	7.09	3.65	216.18	30.00	1.05	0.40	0.00
F13	1.91	1.44	6.86	3.66	296.75	0.00	0.00	0.06	0.00
F14	2.26	1.38	6.96	3.69	269.78	0.00	0.00	0.18	0.00
F15	1.57	1.25	7.07	4.60	322.53	0.00	2.00	0.08	0.00
F16	2.40	1.01	4.10	2.89	207.49	2.37	11.73	2.40	0.00
F17	4.34	0.97	4.51	3.04	256.61	0.00	0.35	0.31	0.00
F18	11.37	0.35	3.58	2.68	197.29	0.00	2.76	0.00	0.00
F19	1.30	0.47	3.92	2.76	195.82	0.00	0.00	0.06	0.00
F20	1.59	1.11	4.94	3.09	203.90	0.00	4.22	0.11	0.00
F21	2.27	2.07	6.14	3.66	214.84	1.06	9.77	3.97	0.00
F22	3.48	3.31	7.53	6.50	227.16	4.61	14.28	14.62	0.00

[a] Results taken from Trafalis and Kasap [24]

[b] Because V5 was tested with different numbers of processes, maximum and minimum CPU time costs are given by V5

We have done a group of experiments on the Beowulf PCs Cluster and are here to randomly choose one of them and list experimental results in Table 18. In terms of these results, it is obvious to note that the total CPU time originally used began to decrease as CPUs were added, but after the number of CPUs reached around 40, there was no significant change in the total CPU time. On the contrary, it began to gradually increase as CPUs were added. In general, the time cost of a parallel program

Table 18 Performance measure data

Num. of CPUs	Program execution time (s)	Total CPU time (s)	Relative speed-up	Relative efficiency
2	5506	10966.13	1	1
4	2572	10240.79	2.14	1.07
8	1011	7965.61	5.45	1.36
16	442	6620.83	12.46	1.56
24	329	6919.24	16.73	1.39
40	208	5494.18	26.47	1.32
80	224	6209.75	24.58	0.62
120	286	6697.64	19.25	0.32
160	362	6569.19	15.21	0.19

Table 19 Total communication and waiting time cost per CPU versus the number of CPUs used

Time (s)	The number of CPUs used									
	2	4	8	10	20	40	80	120	160	192
T_{comm}	0.01	0.01	0.07	0.06	0.11	0.20	0.48	2.01	2.25	4.10
T_{wait}	45.14	115.88	207.11	235.61	455.94	909.96	1715.09	2628.08	3428.22	4619.24

T_{comm} is the total communication time per CPU and T_{wait} is the total waiting time cost per CPU

mainly consists of three parts, total CPU time, total communication time and total waiting time. It is easy to get to know the total communication time that has to be used because the algorithm needs to exchange data periodically. In addition, it is not difficult to know the total waiting time that has to be used because all participating processes in a parallel program need to be synchronized before communication can be done. Here are a group data about communication and waiting time cost per CPU. These results in Table 19 and Fig. 8 were obtained on our IBM Beowulf PCs Cluster. As mentioned before, networking in our IBM Beowulf PCs cluster is accomplished with Myrinet with full-duplex $2+2$ Gigabit/second links. From the experimental data given in Table 19, we can see that the total communication cost per CPU is very small due to high-speed Myrinet networking. The time cost in communication per CPU can almost be ignored compared to total waiting time cost. The above results also reveals that the total communication and waiting time cost per CPU almost linearly increase as the number of CPUs is added. However, the results revealed in Fig. 9 show that the total CPU time cost parabolically changes with the number of CPUs used. Therefore we think this is the cause that results in the effect revealing in Fig. 7.

In the light of the experimental results given in Table 18, Fig. 9 illustrates the relationship between total CPU time and the number of CPUs added, Fig. 10 illustrates the relationship between relative speed-up and the number of CPUs used, and Fig. 11 illustrates the relationship between relative efficiency and the number of CPUs used. Apparently, after the number of CPUs added reaches around 40, there is no available scalability. Simultaneously, relative efficiency is less than one, and this means hardware resources of the high-performance computer are wasted. Therefore, the experimental results revealed here deny the hypothesis that the bigger the number of CPUs used, the better the performances of a parallel program.

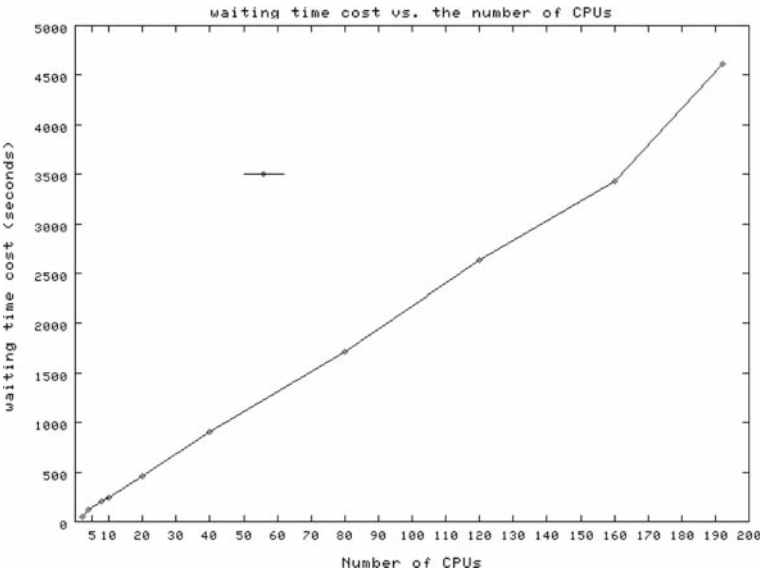

Fig. 8 The relationship between total waiting time cost per CPU and the number of CPUs used

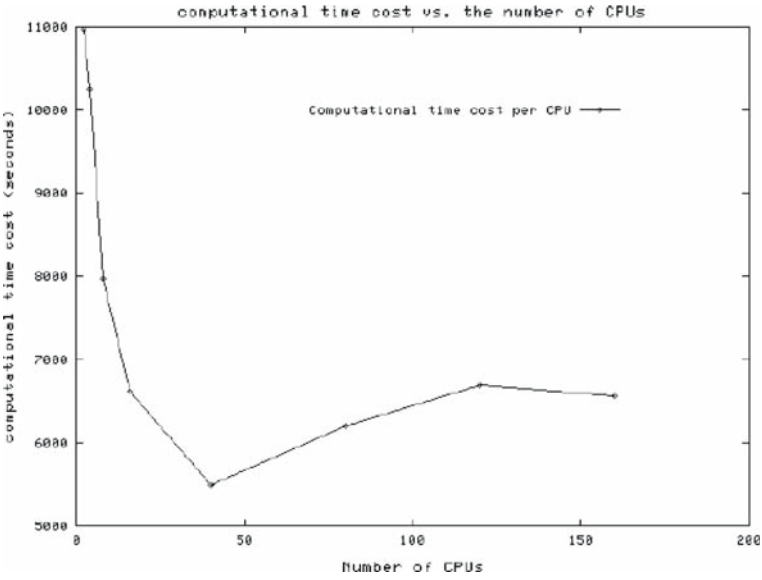

Fig. 9 The relationship between total CPU time and the number of CPUs used

4 Conclusions

The prohibitively long execution time of simulated annealing due to its sequential nature hinders its application to realistically sized problems. A more efficient way to reduce CPU time cost and make the SA a more promising method is to parallelize sequential simulated annealing based on high-performance computing. It is

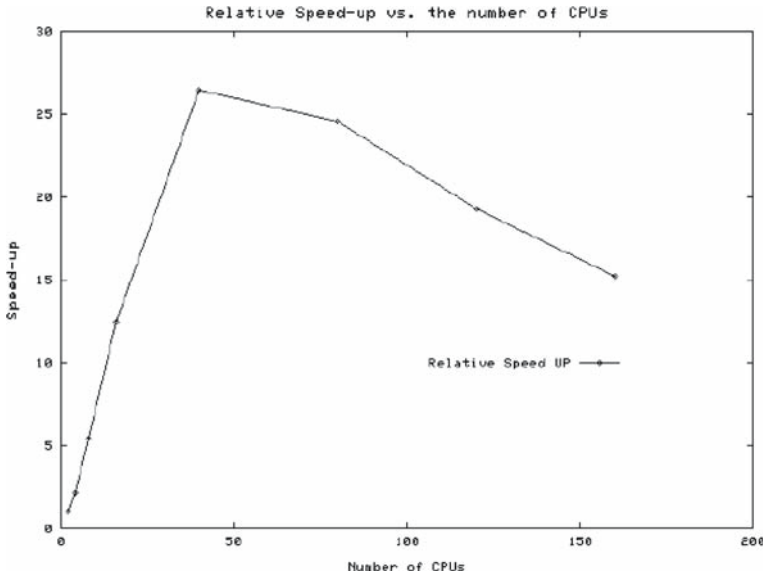

Fig. 10 The relationship between relative speed-up and the number of CPUs used

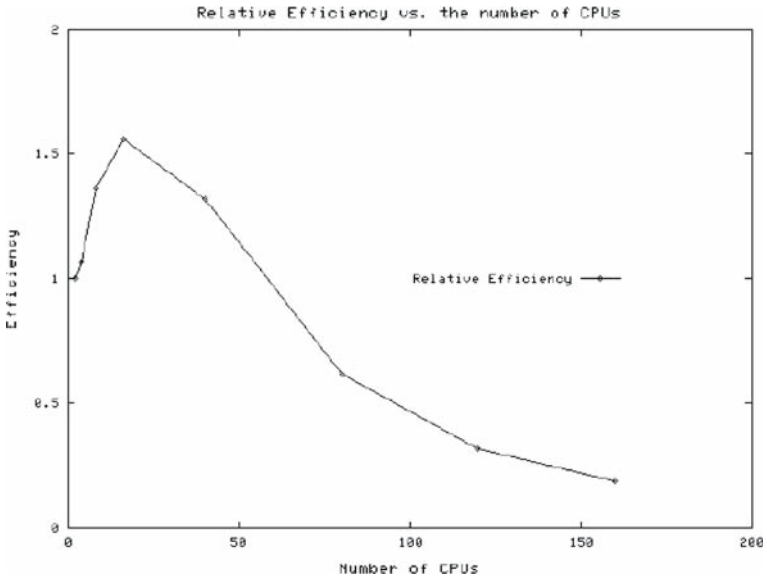

Fig. 11 The relationship between relative efficiency and the number of CPUs used

a challenging task. In fact, there are many approaches that may be considered in parallelizing SA. However, an inappropriate strategy used will likely result in poor performance.

In this paper, we have used five different considerations to do this work. From the experimental results, we eventually found out that the traditional approach to parallelizing simulated annealing, namely parallelizing moves in sequential SA (here V2 and

V3 belong to this situation), had difficulty in handling the very large problem instances. A divide-and-conquer decomposition strategy used for searching space (here V1 belongs to this case) sometimes might find the global minimum, but it leads to long time cost once the random search space was considerably expanded. The most effective way we found to identify the global optimum solution was to introduce genetic algorithm (GA) and build a GA + SA highly hybrid algorithm (here V5 belongs to this case). In this approach, GA was applied to each cooling temperature stage. Some comparisons of the performance of V5 have been made with some recent global optimization algorithms in terms of the number of functional evaluations needed to obtain a global minimum, success rate and solution quality. Results show that V5, the best of the proposed algorithms, outperforms other algorithms used. Additionally, further performance analyses on V5 itself in total CPU time cost, relative speed-up and efficiency amply show that the scalability of a parallel program is limited and the expectation of always being able to increase speedup is not practical. This point seems to suggest that there is no need to increasingly add CPUs. However, in fact, a tradeoff between capacity and efficiency exists. As shown by the experimental results in Sect. 3.2, particularly for those extremely difficult function optimization problems, if the efficiency is not the mainly concerned problem, parallel algorithms running with the bigger number of CPUs potentially yields more opportunities to find or come close to global optimal solutions.

Acknowledgements Authors gratefully acknowledge the constructive criticism and comments of the reviewers that helped in improving the content of this paper notably. In addition, this research work was ever supported by Brain Korea 21 Project, the school of information technology, KAIST in 0000.

Appendix A

Function 1, 2, and 3 (*N-D*)*:* D-dimensional nowhere-differentiable test function: $X_i \in [\pm 1000,\ \pm 1000] \forall i$. This function has infinite number of local minima and one global minimum and the global objective function value is 1.

$$f(x) = \prod_{k=1}^{N} \left(1 + k \sum_{n=0}^{\beta} \frac{|2^n x_k - \lfloor 2^k x_k \rfloor|}{2^n} \right),$$

where $\beta = \infty$. In our test experiments, $\beta = 30$, and $N = 10, 15$, or 20.
Function 4, 5, 6, and 7 (*N-D*)*:* Modified Griewank test function: $X_i \in [\pm 600,\ \pm 600] \forall i$. This function has a very large number of local minima, exponentially increasing with N. It has one global minimum value 0.

$$f(x) = \sum_{i=1}^{N} \frac{x_i^2}{4,000} - \prod_{i=1}^{N} \left(2 + \cos\left(\frac{x_i}{\sqrt{i}} \right) \right) + 3^N.$$

In our test experiments, $N = 2, 4, 6$, or 8.
Function 8 (*2-D*)*:* Goldstein and Price's function: $X_i \in [\pm 2,\ \pm 2] \forall i$. There are four local minima and the global objective function value is 3.

$$f(x) = [1 + (x_1 + x_2 + 1)^2 (19 - 14x_1 + 3x^2 - 14x_2 + 6x_1 x_2 3x_2^2)]$$
$$[30 + (2x_1 - 3x_2)^2 (18 - 32x_1 + 12x_1^2 + 48x_2 - 36x_1 x_2 + 27x_2^2)].$$

Function 9 (*2-D*)*:* Himmelblau's function: $X_i \in [\pm 6,\ \pm 6] \forall i$. There are four global minima with an objective function value of 0.

$$f(x) = \left(x_1^2 + x_2 - 11\right)^2 + \left(x_1 + x_2^2 - 7\right)^2.$$

Function 10 (*2-D*)*:* Shubert function: $X_i \in [\pm 20,\ \pm 20] \forall i$. This function has more than 760 local minima and more than 18 global minima with an objective function value of -186.7309.

$$f(x) = \left\{\sum_{i=1}^{5} i\cos[(i+1)x_1 + i]\right\}\left\{\sum_{i=1}^{5} i\cos[(i+1)x_2 + i]\right\}.$$

Function 11 (*2-D*)*:* General test function: $X_i \in [\pm 5,\ \pm 5] \forall i$. This function has several local minima with an objective function value of 0.

$$f(x) = 0.5x_1^2 + 0.5[1 - \cos(2x_1)] + x_2^2.$$

Function 12, 13,14, and 15 (*2-D*)*:* General test function: $X_i \in [\pm 5,\ \pm 5] \forall i$ for functions 12 and 13, and $X_i \in [\pm 20,\ \pm 20] \forall i$ for functions 14 and 15. These functions have more than three local minima and two global minima.

$$f(x) = 10^n x_1^2 + x_2^2 - (x_1^2 + x_2^2)^2 + 10^m (x_1^2 + x_2^2)^4, n = -m.$$

For $n = 1, n = 2, n = 3$ and $n = 4$, global object function values are -0.407461, -18.058697, 227.765747, and -2429.414749 respectively.

Function 16 (*2-D*)*:* Rastrigin function: $X_i \in [\pm 5,\ \pm 5] \forall i$. This function has 50 local minima and one global minimum with an objective function value of -2.

$$f(x) = x_1^2 + x_2^2 - \cos(18x_1) - \cos(18x_2).$$

Function 17 (*2-D*)*:* Branin function: $X_i \in [\pm 20,\ \pm 20] \forall i$. This function has more than three local and global minima with an objective function value of 0.397887.

$$f(x) = (x_2 - 5.1x_1^2/4\pi^2 + 5x_1/\pi - 6)^2 + 10(1 - 1/(8\pi))\cos x_1 + 10.$$

Function 18 (*1-D*)*:* General test function: $X_i \in [\pm 20,\ \pm 20] \forall i$. This function has 38 local minima and 7 global minima with an objective function value of -3.372897.

$$f(x) = -\left\{\sum_{i=1}^{5} \sin[(i+1)x + i]\right\}.$$

Function 19 (*2-D*)*:* Rosenbrock's function: $X_i \in [\pm 2,\ \pm 2] \forall i$. There is a unique global minimum with an objective function value of 0.

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2.$$

Function 20, 21 and 22(*N-D*)*:* General test function: $X_i \in [\pm 20,\ \pm 20] \forall i$. This function has 2^N local minima and one global minimum. The global objective function values are -78.332331, -117.4984, and -156.66466 for N equal 2,3, and 4, respectively.

$$f(\bar{x}) = (1/2)\sum_{j=1}^{N}\left(x_j^4 - 16x_j^2 + 5x_j\right).$$

References

1. Wah, B.W., Chang, Y.-J.: Trace-based methods for solving nonlinear global optimization problems. J. Global Optimiz. **10**(2), 107–141 (1997)
2. Szu, H., Hartley, R.: Fast simulated annealing. Phys. Lett. A. **122**, 157–162 (1987)
3. Rosen, B.E.: GA's and very fast simulated reannealing. Genetic Algorithms Digest **5**(36), (1991) An Electronic Journal
4. Ingber, L., Rosen, B.: Very fast simulated reannealing (VFSR), netlib@research.att.com:/netlib/opt/vfsr.Z, AT&T Bell labs, Murray Hill, NJ (1992)
5. Ingber, L.: Simulated annealing: practice versus theory. J. Math. Comput. Model **18**(11), 29–57 (1994)
6. Siarry, P., Berthiau, G., Durbin, F., Haussy, J.: Enhanced simulated annealing for globally minimizing functions of many-continuous variables. ACM Trans. Math. Software **23**(2), 209–228 (1997)
7. Ellen, E.W., Roger, D.C., Mark, A.F.: Parallel simulated annealing using speculative computation. IEEE Trans. Parallel Distribut Syst. **2**(4), 483–494 (1991)
8. Hamma, B.S., Viitanen, S., Torn, A.: Parallel continuous simulated annealing for global optimization. Presented at the NATO Advance Study Institute – Algorithms for Continuous Optimization: The State of the Art, II Ciocco-castelvecchio Pascoli, Italy (1993)
9. Yong, L., lishan, K., Evans, D.J.: The annealing evolution algorithm as function optimizer. Parallel Computing **21**(3), 389–400 (1995)
10. Chen, H., Flann, N.S., Watson, D.W.: Parallel genetic simulated annealing: a massively parallel SIMD algorithm. IEEE Trans. Parallel Distribut Syst. **9**, 126–136 (1998)
11. Esin, O., Linet, O.: Parallel simulated annealing algorithms in global optimization. J. Global Optimiz. **19**, 27–50 (2001)
12. Metropolis, N., Rosenbluth, A.W., Rosenbluth, M.N., Teller, A.H., Teller, E.: Equation of state calculations by fast computing machines. J. Chem. Phys. **21**(6), 1087–1092 (1953)
13. Kirkpatrick, S., Gelatt, C.D. Jr., Vecchi, M.P.: Optimization by simulated annealing, science **220**(4598), 671–680 (1983)
14. Kimura, K., Taki, K.: Time-homogeneous parallel annealing algorithm. Report TR-673, Institute for New Generation Computer Technology, Tokyo, Japan (1991)
15. Mahfoud, S.W., Goldberg, D.E.: Parallel recombinative simulated annealing :a genetic algorithm. IlliGAL Report No. 92002, University of Illinois, Urbana, IL (1992)
16. Ter Laak, A., Hertzberger, L.O., Sloot, P.M.A: Nonconvex continuous optimization experiments on a transputer system. In: Allen, A. (ed.) Transputer Systems-Ongoing Research, pp. 251–265. IOS Press, Amsterdam (1992)
17. http://www.mcs.anl.gov/mpi/mpich and http://www.mpi-forum.org
18. Korf, R.E.: Linear-space best-first search. Artif. Intell. **62**, 41–78 (1993)
19. Pardalos, P.M., Rosen, J.B.: Constrained Global Optimization: Algorithms and Applications Vol. 268 of Lecture Notes in Computer Science. Springer-Verlag, Berlin (1987)
20. Parker, R.G., Rardin, R.L.: Discrete Optimization. Academic Press Inc., San Diego, CA (1988)
21. Kennedy, J., Eberhart, R.C.: Swarm intelligence. Morgan Kaufmann Publishers, Los Altos (2001)
22. Torn, A., Zilinskas, A.: Global optimization. Springer-Verlag, Berlin (1989)
23. Cetin, B.C., Barhen, J., Burdick, W.: Terminal Pepeller unconstrained subenergy tunneling (TRUST) for fast global optimization. J. Optimiz. Theory Appl. **77**(1), 97–127 (1993)
24. Theodore, B.T., Suat, K.: A novel metaheuristics approach for continuous global optimization. J. Global Optimiz. **23**, 171–190 (2002)
25. Corana, A., Marchesi, M., Martini, C., Ridella, S.: Minimizing multimodal functions of continuous variables with the simulated annealing algorithm. ACM Trans. Math. Software **13**(3), 262–279 (1987)
26. Cem, B.: A hybrid parallel simulated annealing algorithm to optimize store performance. Proceedings of Workshop on Evolutionary Computing for Optimisation in Industry at the Genetic and Evolutionary Computation Conference (GECCO-2002), 9 July 2002, New York (2002)
27. Hussain, M.F., Al-sultan, K.S.: A hybrid genetic algorithm for nonconvex function minimization. J. Global Optimiz. **11**, 313–324 (1997)
28. Alan, H.K., Horace, P.F.: Measuring parallel processor performance. Commun. ACM. **33**(5), 539–543 (1990)