A Parallel Controlled Random Search Algorithm

Y. F. Hu K. C. F. Maguire R. J. Blake

Daresbury Laboratory, Warrington WA4 4AD, UK

May 6, 1999

Abstract

In this paper a parallel Controlled Random Search (CRS) algorithm is suggested. Unlike the standard CRS algorithms, the parallel algorithm (PCRS) generates more than one new point in parallel each iteration. The parallel algorithm has been tested on a Cray T3D parallel computer, demonstrating reasonablly good scalability for up to 64 processors. The algorithm has also been compared with genetic algorithms.

Keywords: Global optimization, direct search method, controlled random search, genetic algorithm, parallel computing.

1 Introduction

With the increase in the accuracy and speed of computer simulations comes the challenge of using such simulations as a tool for the optimal design of systems and products. There has been a lot of effort in this area, such as shape optimization of airfoils and artificial heart components [10, 21], and optimal design of structures for the reduction of vibrations [12].

1.1 The design optimization problems

Mathematically, many design optimization problems can be viewed as finding the global minimizer x^* of an objective function f over a feasible region $S \subset R^n$. The point $x^* \in S$ is a global minimizer if $f(x^*) \leq f(x)$ for all $x \in S$. A number of global optimization algorithms have been proposed, including deterministic algorithms [11] and stochastic algorithms [4, 5, 14, 22, 23].

There are a few distinctive features of a design optimization problem

- The cost of evaluating the objective function can be very high. This is due to the fact that usually the function is not given analytically, but rather defined by a "black-box" simulation code (such as a CFD code). One complete simulation is therefore required for each function evaluation.

- In many cases the analytical derivatives are not available and numerical derivatives must be used instead. Furthermore, the function value can be subject to noise, making accurate numerical derivatives difficult to calculate.

Taking these features into consideration, global optimization methods that do not use gradient information can sometimes be preferred over the gradient based algorithms [6, 7]. Such methods are known as direct search methods.

1.2 Review of controlled random search algorithms

The controlled random search algorithm for minimization is a direct search method. It is a simple iterative algorithm, typically of the following form

Algorithm CRS

- Step 1. Generate an initial sample G: generate N random points in the

feasible region S. Let x_l and x_h be the points with the smallest and largest function values.

- Step 2. Generate a point: generate a new point $x_{\text{new}} \in S$ based on the information given by the N current sample points.

- Step 3. Contraction: if $f(x_{\text{new}}) \leq f(x_h)$, delete x_h from the sample G and enter x_{new} into G. Update the best and worst points x_l and x_h in the sample.

- Step 4. Repeat from Step 2 until the function values of the points in the sample are "close" to each other.

A number of modifications to the original controlled random search algorithm (CRS) of Price [18] have been proposed [1, 2, 15, 16, 19, 20], the main difference among them lies in the way the new trial point is generated (Step 2). In the original algorithm (CRS), the new trial point is defined by selecting a simplex of $n+1$ points, $\{x_1, x_2, \ldots, x_{n+1}\}$, randomly from the sample of N points, and reflecting x_{n+1} against $\bar{x}$, the center of the simplex. The new point is therefore

$$x_{\text{new}} = 2\bar{x} - x_{n+1}. \tag{1}$$

This was modified in [19] by setting x_1 to be the best point found so far ($x_1 = x_l$) and the resulting algorithm is denoted as CRS2. This algorithm was further modified [20] by employing a Nelder-Mead [17] local search procedure after Step 2, if the new point is found to be better than the current best point. In [15] a Golden Section line search approach was incorporated into the simplex procedure. In [1], a local search procedure based on the β distribution was applied if CRS finds a point that is better than the current best.

Most recently Ali et al. [2] proposed CRS6, a modified controlled random search algorithm. This algorithm used quadratic interpolation of the current

best point x_l, and two other randomly chosen sample points, to generate a new point x_{new}. A local search procedure based on β-distribution was incorporated after Step 2 if x_{new} was better than the current best point x_l. Numerical results showed that CRS6 was considerably more efficient than CRS, with the majority of improvement resulting from the use of quadratic interpolation. The use of quadratic interpolation in the controlled random search algorithm was also proposed in [16].

1.3 Genetic algorithms

Another direct search algorithm which has attracted a lot of interest in recent years is the Genetic Algorithm (GA). This is a member of a more general class of Evolutionary Algorithms, which imitate the process of evolution. In GAs, a point in the feasible region is represented by a string. Typically binary coding is used, but for continuous optimization problems real coding may be preferable. The algorithm starts with a first generation of strings (initial population). Selected pairs of strings "crossover" to produce new offspring. Those strings with higher fitness (smaller function values) have a better chance of being selected and taking part in this crossover process. The resulting offspring form part of the new generation, and this process repeats until the deviation within the population is less than a certain tolerance. During this process strings also have a small probability of being randomly mutated.

It is interesting to relate the controlled random search algorithm to the GA. The sample G of the CRS can be considered as the population and N the population size; in the crossover phase of the genetic algorithm, m new offspring are formed by the crossover of the binary (or floating point) representation of selected parents, while in CRS only one new offspring is generated each iteration, using a

number of strategies including quadratic interpolation of 3 points (as in CRS6, [2]) or the reflection of a vertex of a simplex against the center of the simplex (as in CRS, [18]).

1.4 The case for parallel algorithms

Although direct search algorithms are robust and noise tolerant, they tend to require many more function evaluations to converge to the minimum when compared with gradient based algorithms. For design optimization problems where function values are expensive to evaluate, if the function values are evaluated sequentially, the substantial computation time could prohibit the use of such direct search algorithms. One remedy is to modify the algorithms so that many function evaluations can be carried out at the same time on different processors of a parallel computer.

GAs are intrinsically parallel, because the crossover and the fitness evaluation of the offspring can be performed in parallel. On the other hand, CRS, in its original form, does not have the same parallelism, due to the fact that only one new point is generated and evaluated at each iteration.

The motivation of this paper is to investigate whether parallelism could be introduced into the controlled random search algorithms by allowing multiple offspring to be formed, and whether such parallel algorithms will be efficient.

In Section 2 the parallel CRS algorithm is given together with some implementation details. The algorithm is tested in Section 3 using a set of standard test functions on a Cray T3D parallel computer. The parallel CRS algorithm is also compared with a GA algorithm and is found to be very competitive for the problems considered.

2 The Parallel CRS algorithm

It is assumed that the parallel CRS algorithm, to be detailed later, will run on a parallel computer with p processors ($p \geq 1$). The algorithm is designed for a distributed memory parallel computer, where access to the remote memory is far more expensive than the access to local memory. Test results given later will be on such a machine, the Cray T3D. However the algorithm would work equally well on shared memory parallel computers.

It is also assumed that the evaluation of functions is far more expensive than other computational and communicational costs such as vector-vector operations or random number generation. This assumption is valid for most design optimization problems.

In the rest of the paper some of the terminology from GAs is used, such as *generation* and *offspring*, which stand for *sample* and *new trial points* respectively. The term *generation* is also used interchangeably with *population*.

The parallel controlled random search algorithm, PCRS, is given as follows.

Algorithm PCRS

- Step 1. Generate the initial population G: on each processor, generate N random points in the feasible region S and evaluate the function values of N/p of these points (in parallel). Broadcast the function values so that each processor has a record of the function value of every point in G. Let x_l and x_h be the points with the smallest and largest function values in G.

- Step 2. Generate offspring ("crossover"): on each processor, generate $m \geq 1$ offspring in the feasible region and evaluate their function values. Discard those offspring that are worse than x_h, and broadcast the remaining offspring and their corresponding function values.

- Step 3. Contraction: sort the current population G together with all the new offspring, keep the best N points as the new generation G. Update the best and worst point x_l and x_h.

- Step 4. Repeat from Step 2 until $f(x_h) - f(x_l) < \epsilon$.

As can be seen, in the parallel algorithm the N sample points are replicated over all processors. The initial N random points are generated on all processors by using the same random number generator with the same seed. The function values are evaluated in parallel by each processor working on a portion of the generation. The function values are then broadcasted to all processors.

In the Step 2 of the algorithm, m feasible offspring are generated in parallel on each processor and their function values evaluated. Any of the strategies used in the sequential CRS algorithm to generate a new trial point can be adopted here. Those offspring that are better than the worst point in the current generation are kept. A broadcast will allow each processor to have a copy of all offspring and their function values. The best N of all points will form the new generation. Each processor, when generating the m offspring, uses a different seed for the random number generator to avoid producing the same offspring as other processors.

If R denotes the strategy of generating the offspring, then the above algorithm can be written as PCRS(p, m, R), where p is the number of processors and m the number of offspring per processor. The traditional sequential controlled random search algorithms can be written as CRS=PCRS$(1, 1, R)$.

3 Numerical Results

3.1 "Crossover" strategies

Two "crossover" strategies are used. The first is that of CRS2 [19], as described in Section 1, equation (1). The resulting algorithm is denoted as PCRS(p, m, Δ), where Δ stands for "simplex".

The second "crossover" strategy follows that of [2, 16]. Here the best point $a = x_l$ together with two randomly selected points $\{b,\ c\}$ out of the current generation are chosen. The i-th component of a new offspring is set to be the stationary point of the one-dimensional quadratic that interpolates the three twin-lets $\{a_i, f(a)\}$, $\{b_i, f(b)\}$, $\{c_i, f(c)\}$. Thus

$$(x_{\text{new}})_i = \frac{1}{2}\frac{(b_i^2 - c_i^2)f(a) + (c_i^2 - a_i^2)f(b) + (a_i^2 - b_i^2)f(c)}{(b_i - c_i)f(a) + (c_i - a_i)f(b) + (a_i - b_i)f(c)}.$$

If the denominator is zero (which could happen if the three twin-lets are on a line), or the offspring x_{new} is outside the feasible region, then replace b and c with another two randomly chosen points in the current generation G and restart the process. The resulting algorithm is denoted as PCRS(p, m, q), where q stands for "quadratic".

As far as we understand, there is no theoretical justification for the use of such a 1-dimensional quadratic interpolation on a multidimensional problem. However the numerical results in [2, 16] clearly demonstrate that this "crossover" strategy can result in a more efficient algorithm than that based on the use of simplex. We observed, through numerical experiment on a multi-dimensional quadratic function, that the offspring generated by the 1-D quadratic interpolation tend to be randomly distributed, with a high concentration towards the neighborhood of the minima of the multi-dimensional quadratic.

Table 1: The Test Problems

Function	n	bound	f^*
Branin (BR)	2	$-5 \le x_1 \le 10$	0.3978
		$0 \le x_2 \le 15$	
Goldprice(GP)	2	$-5 \le x_i \le 10$	3.0000
Shekel5 (S5)	4	$0 \le x_i \le 10$	-10.1532
Shekel7 (S7)	4	$0 \le x_i \le 10$	-10.4029
Shekel10 (S10)	4	$0 \le x_i \le 10$	-10.5364
Hartman3 (H3)	3	$0 \le x_i \le 1$	-3.8627
Hartman6 (H6)	6	$0 \le x_i \le 1$	-3.3223
Schubert3 (P8)	3	$-10 \le x_i \le 10$	0.0000
Schubert5 (P16)	3	$-5 \le x_i \le 5$	0.0000
Levy10 (L10)	10	$-10 \le x_i \le 10$	0.0000
Kowalik (KL)	4	$0 \le x_i \le 0.42$	0.0003
Hosaki (HK)	2	$0 \le x_1 \le 5$	-2.3460
		$0 \le x_2 \le 6$	
Powell(PW)	4	$-10 \le x_i \le 10$	0.0000

3.2 Numerical results

The two algorithms were implemented in FORTRAN 77 using MPI [24], the message passing standard. They were tested on a set of 13 test functions, using up to 256 processors of a Cray T3D parallel computer. For this experiment the number of offspring on each processor is restricted to one. The size of the population N is set to

$$N = 10\ (n+1),$$

where n is the number of variables.

Table 1 gives some details of the set of 13 test functions used. Further details of these test problems can be found [2]. Each test problem involves the minimization of a function, subject to simple bound constraints. Each algorithm was run 100 times on each of the test functions to even out the fluctuations caused by the use of random numbers. Each run is terminated when the difference in the function values between the best and worst points in the population is less than $\epsilon = 10^{-4}$.

Table 2: The number of function evaluations of the parallel controlled random search algorithm PCRS$(p, 1, q)$ (based on quadratic interpolation) on 13 test functions.

$p =$	1	2	4	8	16	32	64	128	256
BR	218	111	57	31	16	11	10	9	9
GP	182	106	46	25	15	10	9	8	8
S5	845	416	216	118	60	33	23	25	23
S7	827	405	208	106	57	32	24	22	20
S10	855	401	213	107	60	32	23	22	22
H3	257	133	68	37	20	12	10	9	9
H6	927	478	255	128	66	38	23	20	18
P8	261	132	70	37	20	12	10	9	9
P16	601	303	156	80	41	25	15	14	13
L10	1731	877	443	224	118	65	38	26	23
KL	198	101	53	27	15	8	6	6	5
HK	148	77	40	21	11	8	7	7	7
PW	1631	769	404	201	101	53	27	27	19
Average	667.77	331.46	171.46	87.85	46.15	26.08	17.31	15.69	14.23
Speedup	1.00	2.01	3.89	7.60	14.47	25.61	38.58	42.55	46.92

A run was assumed to have failed, if the minimum function value found had a relative error of more than 0.1% compared with the known global minimum, or if the number of function evaluations on any processor exceeds 10000.

The results of the parallel controlled random search algorithms PCRS$(p, 1, q)$ and PCRS$(p, 1, \Delta)$ are given as follows. In Table 2 and Table 3, the average number of function evaluations per processor for the successful runs are listed. The average number of function evaluations are given in the row labelled by "Average", which gives an indication of the average efficiency of the algorithm. The speedups are reported in the last row. This is defined as the average number of function evaluations on one processor for the sequential algorithms PCRS$(1, 1, R)$, divided by the average number of function evaluations for the corresponding parallel algorithms PCRS$(p, 1, R)$ $(p > 1)$. The speedups are also plotted against

Table 3: The number of function evaluations of the parallel controlled random search algorithm PCRS$(p, 1, \Delta)$ (based on simplex) on 13 test functions.

$p =$	1	2	4	8	16	32	64	128	256
BR	518	261	138	69	39	23	25	25	25
GP	680	346	174	91	46	29	30	30	30
S5	3293	1650	845	426	215	116	80	79	78
S7	3008	1525	768	391	201	109	74	72	71
S10	3080	1603	793	400	209	112	76	74	73
H3	917	459	234	121	63	35	29	28	28
H6	4014	2101	1031	541	271	144	79	74	75
P8	1369	701	344	180	94	51	42	41	40
P16	3375	1703	850	429	222	115	68	66	65
L10	8963	4512	2259	1153	586	307	167	110	114
KL	528	268	135	69	36	19	14	14	14
HK	476	240	125	63	33	20	21	20	20
PW	1957	984	499	253	132	70	49	48	47
Average	2475.23	1257.92	630.38	322.00	165.15	88.46	58.00	52.38	52.30
Speedup	1.00	1.97	3.93	7.69	14.99	27.98	42.68	47.25	47.32

the number of processors in Figure 1.

In Figure 2, the reliability of the two algorithms is plotted against the number of processors. The reliability is defined as the average percentage of successful runs over 13 test functions.

From Figure 1 and Tables 2 and 3, it is seen that in terms of the number of function evaluations, the parallel algorithms scale reasonably well against the number of processors for up to 64 processors. For example, from Table 2, with 64 processors, algorithm PCRS(64,1,q) takes an average of 17.31 function evaluations. Compared with 667.77 function evaluations for the sequential algorithm PCRS(1,1,q), this represents a speedup of 38.58. Beyond 64 processors, the improvement in the speedup tails off, as is evident from Figure 1. This is because, for instance, the PCRS$(64, 1, q)$ requires on average only 17 function evaluations

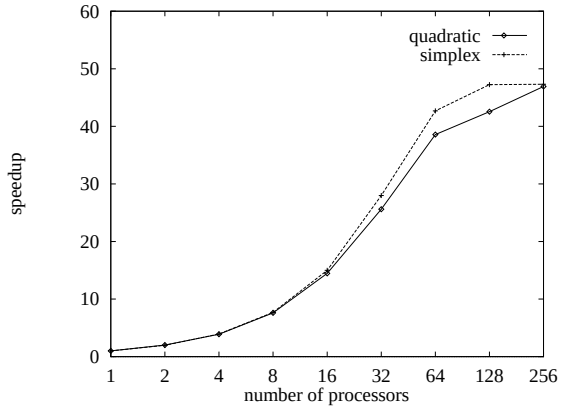

Figure 1: The speedup of parallel controlled random search algorithms PCRS$(p, 1, q)$ (quadratic) and PCRS$(p, 1, \Delta)$ (simplex), against the number of processors

(see Table 2). This is equivalent to about 16 iterations (generations). On other the test problems the algorithm sometimes converges in 6-10 iterations. At this level it is very difficult to improve further. On those functions which require more function evaluations, there is still ground for improvement. For example on test problem L10 the number of function evaluation is reduced from 38 on 64 processors to 26 on 128 processors.

The timings of all the experiments are not reported. This is because all the test functions here are relatively cheap to evaluate. For practical design optimization problems, the cost of the function evaluation is likely to be much greater than that for the test functions used here, while sequential overhead and communication cost per iteration of the algorithm will stay constant. The number of function

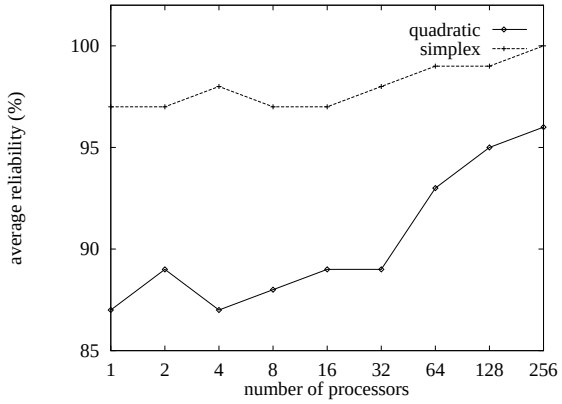

Figure 2: The reliability of the parallel controlled random search algorithms PCRS$(p, 1, q)$ (quadratic) and PCRS$(p, 1, \Delta)$ (simplex), against the number of processors

evaluations is therefore a more important measure of the parallel scalability.

Comparing PCRS$(p, 1, \Delta)$ and PCRS$(p, 1, q)$, which use simplex and quadratic interpolation respectively as the "crossover" strategies, it is seen that the former requires three to four times the number of function evaluations. In terms of reliability (see Figure 2), PCRS$(p, 1, \Delta)$ is more reliable than PCRS$(p, 1, q)$, with an average reliability of over 97%. In general, the reliability of both algorithms improves as the number of processors increases. This is because the sequential algorithm PCRS$(1, 1, R)$ generates one offspring each iteration. It then discards the worst point in the current generation if the new offspring is found to be better. In comparison, the parallel algorithm PCRS$(p, 1, R)$ $(p > 1)$ generates p offspring over p processors, before comparing them with points in the current generation.

Table 4: Effect of different number of new points per iteration on a sequential controlled random search algorithm: reliability and average number of function evaluations (NF) over 13 test functions.

Algorithms	reliability%	Average NF	Average NF/p
PCRS$(1, 1, q)$	87	667.77	667.77
PCRS$(1, 2, q)$	88	678.77	339.11
PCRS$(1, 4, q)$	88	688.85	172.21
PCRS$(1, 8, q)$	89	716.38	77.05
PCRS$(1, 16, q)$	90	752.54	47.03
PCRS$(1, 32, q)$	90	831.69	25.99
PCRS$(1, 64, q)$	91	974.00	15.22

The parallel algorithm therefore gives the less fitted "parents" more chance of taking part in the "crossover", thus allowing more possibilities to be explored. This increases the likelihood of finding the global minimum.

Note that if the same sequence of random numbers is used, then the parallel algorithm PCRS$(p, 1, R)$ is mathematically equivalent to the sequential algorithm PCRS$(1, p, R)$ which generates p offspring each iteration, in the sense that both should take the same number of iterations. Thus it is interesting to compare the sequential algorithms PCRS$(1, p, R)$, which generates p offspring per iteration, with the traditional controlled random search algorithms CRS=PCRS$(1, 1, R)$. Table 4 lists the reliability and the average number of function evaluations on the set of 13 test functions, for PCRS$(1, p, q)$ with p ranging from 1 to 64. As can be seen the reliability generally increases with the number of offspring generated at each iteration. The number of function evaluations (NF) increases slowly. This implies that the number of iterations, which is roughly $(NF - N)/p$, decreases as the number of offspring per iteration increases. The results compares well withTable 2.

The speedup in Figure 1 and Tables 2 and 3 is defined as the number

of function evaluations of a traditional controlled random search algorithm CRS=PCRS$(1, 1, R)$, divided by that of the parallel algorithm PCRS$(p, 1, R)$. An alternative definition would be to divide the number of function evaluations of a sequential controlled random search algorithm PCRS$(1, p, R)$ and that of the mathematically equivalent parallel algorithm PCRS$(p, 1, R)$. We believe that the former definition of speedup is more appropriate since PCRS$(1, 1, R)$ is the traditional controlled random search algorithm that we seek to parallelise. From Table 4, PCRS$(1, 1, R)$ also took less function evaluations that PCRS$(1, p, R)$, thus the speedup figures for the parallel algorithm calculated using PCRS$(1, 1, R)$ are more realistic.

4 Comparison with Genetic Algorithms

As discussed in Section 2 the controlled random search algorithms and genetic algorithms are closely related. It is therefore interesting to compare them on the same set of test functions. A typical genetic algorithm for minimization is as follows

Algorithm GA

- Step 1. Generate an initial population G of size N.
- Step 2. Generate offspring
  - Selection: selecting $m \leq N$ strings as parents, with the probability of each string being selected proportional to its fitness (which is inversely proportional to its function value).
  - Crossover: the parents are paired and generate m offspring, which replace the m least fittest strings.

– Mutation: each string has a small probability β of being mutated.

- Step 3. Repeat Step 2 until the function values of the points in G are close to each other.

There are many variants of the above representation of GA. For example, in the crossover phase of Step 2, the m offspring may replace their parents rather than m least fittest strings.

Traditionally, strings are represented by binary numbers and crossover is carried out by swapping parts of the strings between parents. However there were suggestions (see, e.g., [13]) that floating point representation is more natural and efficient for continuous optimization problems. In a floating point representation scheme, crossover of parents $x = (x_1, x_2, \ldots, x_n)$ and $y = (y_1, y_2, \ldots, y_n)$ is carried out by either swapping floating point numbers, analogous to the binary case, or using the arithmetical crossover

$$x_i' = \alpha_i x_i + (1 - \alpha_i) y_i, \quad y_i' = \alpha_i y_i + (1 - \alpha_i) x_i, \quad i = 1, 2, \ldots, n, \tag{2}$$

with α_i uniform random numbers in, say, $[-0.5, 1.5]$.

We initially tested a binary coded GA package on the set of test problems but the results were rather disappointing. In view of the finding in [13] that a floating point represented GA was more efficient than a binary represented one on continuous optimization problems, a floating point coded GA was therefore implemented using the crossover operator (2). The code is sequential and is in FORTRAN 90. In our implementation of GA, the mutation rate is set to $\beta = 0.01$. The mutation for a string x is carried out by setting

$$x_i = x_i + \gamma(XU_i - XL_i),$$

for a randomly selected index i. Here XU_i and XL_i are the upper and lower bound of the element x_i , and γ is a random number between 0 and 1.

Table 5: The number of function evaluations of the parallel GA on 13 test functions.

$p =$	32	64	128
BR	41	37	32
GP	42	36	31
S5	238	129	86
S7	200	112	78
S10	196	107	78
H3	46	31	26
H6	185	142	110
P8	53	41	38
P16	133	74	57
L10	471	366	264
KL	5	5	5
HK	17	14	11
PW	1536	183	60
Average	239.31	94.23	63.38

An important factor in GA is the selection scheme used. Different selection schemes have different "selection pressure" [9], thus different rates of convergence. A number of selection strategies, including Stochastic Universal Sampling (SUS) [3] and tournament selection have been experimented. It was found that the tournament selection with 3 players performed the best.

The number of offspring, m, to be generated per iteration is found to affect the performance of the GA. In general, smaller m results in less overall number of function evaluations for convergence. However a small m also reduces the parallelism of the GA, since the evaluation of the m new offspring can be done in parallel. As a compromise m is set to be $0.25 * N$ and parallelism is adjusted by changing the population size N.

Table 5 give the number of function evaluations of the GA with the population size of $N = 128,\ 256$ and 512, which resulted in $m = 32,\ 64$ and 128 strings being replaced per iteration. On a parallel computer with more than m processors, the

Table 6: Comparing controlled random search algorithms with GA: the average reliability and the average number of function evaluations per processor.

	PCRS$(p,1,q)$		PCRS$(p,1,\Delta)$		GA	
p	reliability%	Average NF	reliability%	Average NF	reliability%	Average NF
32	89	26.08	98	88.46	93	239.31
64	93	17.31	99	58.00	98	94.23
128	95	15.69	99	52.38	99	63.38

m new strings can be evaluated in parallel, the number of function evaluations per processor would therefore be equal to the number of iterations IT, plus $N/m = 4$ for the parallel evaluation of the initial population. Reported in the tables are thus reliability and the projected number of function evaluations NF (=IT+4) per processor. The results are averaged over 100 runs on a DEC workstation. Table 6 summarizes the results for GA and the two PCRS algorithms, by listing the reliability, and the average number of function evaluations per processor per test problem.

From Table 6, the reliability of the two algorithms to the right are seen to be similar and both are more reliable than PCRS$(p,1,q)$, although PCRS$(p,1,q)$ is more efficient. Comparing Table 5 with the corresponding Table 3, it is interesting to see that the number of function evaluations per processor taken by the GA correlates well, qualitatively, to that of PCRS$(p,1,\Delta)$, with the latter being more efficient. The controlled random search algorithms are indeed closely related to the genetic algorithms. The differences lie, in the terminology of genetic algorithms, in the three operators, selection, crossover and mutation, which are listed in Table 7.

The PCRS is more efficient, due to a number of possible reasons. In PCRS, the best string is always selected to take part in the crossover of every offspring. Furthermore, the crossover operators for PCRS are very different from those for

Table 7: Different operators for GA and PCRS

operator	Parallel Controlled Random Search	Genetic Algorithms
Selection	best point always selected, other points randomly selected	tournament; SUS etc
Crossover	simplex; quadratic interpolation	swapping digits; arithmetical
Mutation	no	yes

the GA. In particular the crossover operator of PCRS(p, m, q) is based on finding the local minimums using quadratic interpolation and therefore offers better local convergence, at the cost of reliability.

5 Discussion

In this paper a parallel controlled random search method is suggested. It is found that the parallel algorithms scale reasonablly well against the number of processors, for up to 64 processors. For one of the algorithms considered (PCRS$(p, 1, q)$), the average number of function evaluations needed per processor to solve the 13 test problems is reduced from 667.77 on one processor to 17.31 on 64 processors, giving a speedup of 38.6. This significant reduction in the number of function evaluations makes more realistic the application of controlled random search algorithms to design optimization problem.

The parallel controlled random search algorithm is also compared with the genetic algorithm. In general the PCRS algorithms are shown to be more efficient than a floating point coded genetic algorithm on the set of test problems.

The controlled random search algorithm is strongly related to the genetic algorithm. It would be interesting to explore each algorithm by borrowing operators from the other. For instance, it would be interesting to use a tournament

selection strategy in the PCRS$(p, 1, q)$, and to include a mutation phase, in the hope of improving its reliability. It is also planned to extend the algorithms to handle problems with general nonlinear constraints.

Acknowledgments

The authors would like to thank the referees for their constructive reviews of the paper.

References

[1] M. M. Ali and C. Storey, Modified controlled random search algorithms, *International Journal of Computational Mathematics* **53** (1994) 229-235.

[2] M. M. Ali, A. Törn and S. Viitanen, A numerical comparison of some modified controlled random search algorithms, *Journal of Global Optimization* **11** (1997) 377-385.

[3] J. E. Baker, Reducing bias and inefficiency in the selection algorithms, in J. J. Grefenstette ed., *Proceeding of the Second International Conference on Genetic Algorithms*, (Lawrence Erlbaum Associates, Hillsdale, NJ. 1987), 14-21.

[4] G. Boender, A. Rinnooy Kan, L. Stougie and G. Timmer, A stochastic method for global optimizaion, *Mathematical Programming* **22** (1982) 125-140.

[5] L. De Biase and F. Frontini, A stochastic method for global optimization: its structure and numerical performance, in L. C. W. Dixon and Szeö, G. P. eds., *Towards Global Optimisation 2* (North Holland Publishing Company, 1978).

[6] R. Fletcher, *Practical Methods of Optimization* (John Willey & Sons, Chichester, 1987).

[7] P. E. Gill, W. Murray and M. H. Wright, *Practical Optimization* (Academic Press, London, 1981).

[8] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning* (Addison Wesley, 1989).

[9] P. J. B. Hancock, An empirical comparison of selection methods in evolutionary algorithms, in T. C. Fogarty ed., *Lecture Notes in Computer Science* 865 (Springer-Verlag, 1994) 80-94.

[10] O. Hhattas, B. He and J. F. Antaki, Shape optimization of Navier-Stokes flows with application to optimal design of articicial heart component, EDRC12-67-95. The Engineering Design Research Center, Carnegie Mellon University, Pittsburgh, PA., 1995.

[11] R. Horst and H. Tuy, *Global Optimization (Deterministic Approaches)* (Springer-Verlag, Berlin, 1990).

[12] A. J. Keane, Experiences with optimizers in structural design, Adaptive Computing in Engineering Design and Control, Plymouth, UK, September, 1994.

[13] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs* (Sringer-Verlag, 1996).

[14] J. Mockus, V. Tiešis and Ă. Zilinskas, The application of Bayesian methods for seeking the extremum, in L. C. W. Dixon and G. P. Szeö eds., *Towards Global Optimisation 2* (North Holland Publishing Company, 1978).

[15] C. Mohan and K. Shanker, A numerical study of some modified versions of controlled random search method for global optimization, *International Journal of Computer Mathematics* **23** (1988) 325-341.

[16] C. Mohan and K. Shanker, A controlled random search technique for global optimization using quadratic approximation, *Asia-Pacific Journal of Operational Research* **11** (1994) 93-101.

[17] J. A. Nelder and R. Mead, A simplex method for function minimization, *Computer Journal* **7** (1965) 308-313.

[18] W. L. Price, A controlled random search procedure for global optimisation, in L. C. W. Dixon and G. P. Szeö eds., *Towards Global Optimisation 2* (North Holland Publishing Company, 1978).

[19] W. L. Price, Global optimization by controlled random search, *Journal of Optimization Theory and Applications* **55** (1983) 333-348.

[20] W. L. Price, Global optimization algorithms for a CAD workstation, *Journal of Optimization Theory and Applications* **55** (1987) 133-146.

[21] J. Reuther and A. Jameson, Control theory based airfoil design for potential flow and a finite volume discretization, *AIAA*-94-0499 (1994).

[22] A. Törn, A search clustering approach to global optimization, in L. C. W. Dixon and G. P. Szeö eds., *Towards Global Optimisation 2* (North Holland Publishing Company, 1978).

[23] A. Törn, *Global Optimization*, Lecture notes in computer science, Vol. 350 (Springer-Verlag, 1989).

[24] MPI - a message-passing interface standard, special issue, International Journal of Supercomputer Applications and High Performance Computing, 1994, Vol. 8, No. 3-4. p. 165 et seq.