Acta Cybernetica 14 (1999) 403-418.

A parallelized sequential random search global optimization algorithm

P.M. Ortigosa, J. Balogh, I. García [*]

Abstract

This work deals with a stochastic global optimization algorithm, called CRS (Controlled Random Search), which originally was devised as a sequential algorithm. Our work is intended to analyze the degree of parallelism that can be introduced into CRS and to propose a new refined parallel CRS algorithm (RPCRS). As a first stage, evaluations of RPCRS were carried out by simulating parallel implementations. The degree of parallelism of RPCRS is controlled by a user given parameter whose value must be tuned to the size of the parallel computer system. It will be shown that the greater the degree of parallelism is the better the performance of the sequential and parallel executions are.

Keywords: Parallel algorithm, Distributed Processing, Random Search, Global Optimization.

1 Introduction

The generic global optimization problem can be described as:

$$\min f(s), \;\; s \in S \subset R^n \tag{1}$$

where the objective function, $f(s)$, is a real valued continuous nonlinear function on S and the search domain, S, is a compact body. Under these conditions it is known that the optimal solution value:

$$f^* \equiv \min f(s), s \in S \tag{2}$$

exists and is attained; i.e. the set:

$$S^* \equiv \{s \in S : f(s) = f^*\} \neq \emptyset \tag{3}$$

Two general models of Global Optimization methods exist: Deterministic methods which require a certain mathematical structure and Stochastic methods which

[*]This work was supported by the Tempus Program, by the Consejeria de Educación de la Junta de Andalucía (07/FSC/MDM) and by the Ministry of Education of Spain (CICYTTIC96-1125-C03-03).

are based on the random generation of feasible trial points and nonlinear local optimization procedures. A profound discussion on the classification of methods can be found in Törn and Zilinskas [15] and for a complete and rigorous mathematical description of global optimization methods, both deterministic and stochastic approaches, the reader may consult the Handbook of Global Optimization [6].

On the other hand, there exists a question which frequently arises when a practical Global Optimization problem has to be solved: *which kind of method, or which particular algorithm, may be more appropriate to solve a particular problem?*. The answer could only be obtained from a deep analysis of the problem at hand. A wide discussion on this subject can be found in [5], where relations between the problem, its modeling and properties, and global optimization methods are studied. Roughly speaking, it could be said that deterministic methods may be more efficient than the stochastic ones, when an analytical expression of f, its derivatives, bounds and useful properties are available. However, when f is a black-box function deterministic methods cannot be applied. In contrast, stochastic methods do not require any specific structure of f, only a computational procedure to obtain the value of the function at any location $s \in S$ [5] is needed. So, most optimization problems can be solved by stochastic global optimization techniques.

For computationally expensive functions, stochastic global optimization methods have shown to be very useful because of, compared to deterministic methods, fewer function evaluations are needed to obtain the solution of (1). In addition, stochastic methods can be applied to problems where the objective function is not continuous nor differentiable and only a tool for evaluating the function at any location is required.

For most of the functions, global optimization is a NP hard problem. For this reason, the global optimization problem is a suitable candidate for the use of supercomputers, mainly for those functions whose evaluation is computationally expensive.

This paper will only deal with a stochastic global optimization algorithm called CRS and a new parallel version of CRS. CRS (Controlled Random Search) algorithm was introduced by Price [9, 10, 11]. It is based on clustering techniques and has proved to be very reliable and computationally inexpensive. In [2] a parallel version of CRS (PCRS) was applied to efficiently solve a global optimization problem coming from the image processing field, whose objective function were computationally very expensive.

The aim of this work is to describe and evaluate a new refined version of PCRS, called RPCRS. It was originally devised to be executed on parallel multicomputer system, but it will be also shown that RPCRS outperforms CRS even when it is run on a single processor system. Our study only covers analysis of the speed up of RPCRS as compared to the original sequential CRS algorithm. Our analysis is only based on empirical results obtained from experimental executions. Although a wide set of standard test functions was used to validate our results, this work does not provide any theoretical support to demonstrate that the same results can be obtained using other functions.

Two different kinds of experiments were carried out for analyzing the speed up of RPCRS: those oriented to highlight advantages of its parallel nature and those intended to show its capability for being executed on a parallel computer system.

This paper has been organized as follows: Section 2 describes the CRS algorithm and its parallel version RPCRS. Section 3 is devoted to show experimental results intended to evaluate the speed up of RPCRS compared to CRS, as a function of a control parameter which determines the degree of parallelism. Finally, in Section 4, numerical results of parallel executions of RPCRS, on a CRAY T3D using up to 16 processor elements, will be shown.

2 RPCRS, a parallel version of the CRS algorithm

The goal of this section is to describe the RPCRS algorithm; a refined version of PCRS (Parallel Controlled Random Search) algorithm proposed in [2]. RPCRS is a parallel algorithm based on the Controlled Random Search (CRS) algorithm of Price [9, 10, 11]. Some parallel approaches of the original CRS algorithm have been proposed and evaluated using several models of parallel computers and strategies [1, 3, 4, 7, 12, 14, 16]. Our proposal only makes small modifications to the original sequential version of CRS. These modifications are aimed at increasing the degree of parallelism of CRS by creating a pile of work to feed the set of processors of a parallel computer. RPCRS will allow to evaluate the objective function at several trial points, simultaneously. Nevertheless, the general strategy used in CRS remains in our parallel version. For the sake of clarity, description of CRS will precede to RPCRS algorithm.

The Controlled Random Search algorithm, proposed by Price, is a simple and direct procedure for global optimization, applicable both to unconstrained and constrained optimization problems [9, 10, 11]. In this work, it is assumed that the global optimization problem to be solved is that described by (1), where S is a hyper-rectangle. Due to its simplicity, CRS has been used to solve many practical problems but it has not been very popular among researchers on the theory of Global Optimization because no analytical property can be derived.

CRS starts by evaluating the objective function at N trial points randomly chosen over S, (**initialize** step of Algorithm 2.1). Coordinates and the corresponding value of the objective function, for the set of N trial points, are stored in an array $R = R^0, \ldots, R^N$. The worst and besttrial points in R (R^W, R^B) are then computed at the **update** step. New trial points ($\overline{P}$) are selected and evaluated, at the **generate** step. The algorithm iteratively executes the **update** and **generate** steps until stopping criteria are reached.

At the **generate** step two different trial points are computed; primary and secondary trial points. Both kinds of trial points are defined in terms of the configuration of a subset of $n+1$ ($R^{j_0}, \ldots, R^{j_n}$) trial points. R^{j_i}, ($i = 0, \ldots, n$) are randomly selected from the current set of N points stored at R (CRS is considered the first algorithm which uses a population of points). Primary points are generated in a Nelder-Mead fashion [8] by mirroring a point (R^{j_n}) over the centroid, $\overline{G}$, of the remaining subset of points ($R^{j_0}, \ldots, R^{j_{n-1}}$). In contrast, location of a sec-

Algorithm 2.1

```
Begin CRS($f, S, N, NF_{max}, \epsilon$)
  initialize:
    $Iter = 0;\ D_{max} = \epsilon + 0.1;\ ns = 0$
    Select at random a set, $R$, of $N$ trial points $R^i$; $(0 \leq i < N)$
    Compute $f(R^i)$; $(0 \leq i < N)$
  while ($D_{max} > \epsilon$ OR $f(R^W) - f(R^B) < \delta$ OR $Iter < NF_{max}$)
    update: Determine the trial points $W$ and $B$ such that
              $f(R^W) \geq f(R^i) \geq f(R^B)$; $(0 \leq i < N)$
    $Iter = Iter + 1$
    Begin generate:
      Select randomly $n + 1$ points, $(R^{j_i})$, from the set $R$
      $\overline{G} = \sum_{i=0}^{n-1} R^{j_i}/n$
      $\overline{P} = 2 \times \overline{G} - R^{j_n}$.                # Primary points
      if $\overline{P} \in S$ AND $f(\overline{P}) < f(R^W)$
        $R^W = \overline{P};\ ns = ns + 1$
      else if $RS = ns/Iter < 0.5$                  # Rate of Success Test
        $\overline{P} = (\overline{G} + R^{j_n})/2$                # Secondary points
        $Iter = Iter + 1$
        if $f(\overline{P}) < f(R^W)$
          $R^W = \overline{P};\ ns = ns + 1$
    End generate
  End while
  Return $\{R^B$ and $f(R^B)\}$;                  # $f(R^B) \leq f(R^i)$; $(0 \leq i < N)$
End CRS
```

ondary point is the middle point between R^{j_n} and the centroid $\overline{G}$. While primary points are intended to keep the search space as wide as possible (global search), secondary points are conducive to convergence (local search). Secondary points are only computed if the current primary trial point fails and the rate of success (RS) in finding smaller values of R^W is bellow 50%. This general procedure may be modified in a variety of ways, our version of CRS is detailed at Algorithm 2.1, where stopping criteria are based on (i) the value of the maximum distance between any two points in the set R ($D_{max} = \max\{d(R^i, R^j);\ \forall\ 0 \leq i, j < N;\ i \neq j\} \leq \epsilon$), (ii) the range of $f(R^i)$, i.e. $f(R^W) - f(R^B) < \delta$; where $f(R^W) = \max\{f(R^i)\}$; $f(R^B) = \min\{f(R^i)\}$; and (iii) the number of function evaluations NF_{max}. Condition (i) ensures that all the trial points are located in a small cluster, condition (ii) allows the algorithm to stop even when the trial points are a long distance apart but the values of the function for all the trial points are almost equal one to each other; this condition is useful for functions with several global optima. Finally, condition (iii) will permit to leave the process in the case that algorithm does not converge.

It can be seen that CRS is a highly sequential algorithm, because every new trial point, $\overline{P}$, is generated from a subset of the current set R of N trial points. This current set of points consists of the best points found along the iterative procedure. However, provided that $R^0, \ldots, R^{N-1}$ can be simultaneously generated and $f(R^0), \ldots, f(R^{N-1})$ concurrently computed, the algorithm exhibits some degree of parallelism at the **initialize** stage.

In order to increase the degree of parallelism of CRS, the following strategy has been introduced: After the **initialize** stage, using the same initial set (R) of N trial points, a set of b primary points are generated and saved into a FIFO buffer, $A = A^0, \ldots, A^{b-1}$. After computing $f(A^0)$, R^W will be replaced by A^0 iff $f(A^0) < f(R^W)$. A^0 is removed from the buffer and a new point is obtained by the **generate** procedure and saved at the end of the buffer FIFO. The only difference with Algorithm 2.1 is that a set of b trial points is always ready to be evaluated, so the degree of parallelism is increased.

The best strategy for implementing this kind of parallel algorithms is a centralized model, where a master-worker communication scheme is applied. In our model, the master processor executes the optimization algorithm and provides a set of trial points to the worker processors. worker processors only evaluate the objective function at the trial points supplied by the master processor and after every evaluation of the function they send the result back to the master processor [4]. García et al [2, 4] have implemented a similar strategy using a fully asynchronous model where the master processor does not start to generate primary or secondary trial points until the initial sample set of trial points has been evaluated. At any time worker processors keep information of a single trial point. Although this approach is fully asynchronous, several worker processors frequently may remain in a idle state waiting for the master processor to provide a new trial point.

This drawback could be solved if worker processors always keep in a buffer a set of trial points to be evaluated. So, when a worker processor finishes an evaluation, it sends the result back to the master and goes on evaluating a new trial point stored in its local buffer. Our parallel implementation of CRS (RPCRS) consists of two different processes: **Master_RPCRS** (Algorithm 2.2) and **Worker_RPCRS** (Algorithm 2.3).

The **Master_RPCRS** process consists of three different stages: (i) the **initialize** stage where the set R of N trial points are randomly chosen over S, (ii) a stage where $b = NP \times npoints$ primary trial points are computed and (iii) the convergence loop where primary or secondary trial points are generated following a strategy similar to Algorithm 2.1.

After **initialize** step, master processor cyclically distributes all the N trial points among worker processors. If the number of worker processors (NP) were greater than N, master processor would generate NP trial points and after receiving their function values from worker processors, it would only save the best N trial points at R. Then, master processor calculates b primary trial points and sends *npoints* to each worker processor. Points $\overline{P}$ are computed using b different $\overline{G}$ and R^{j_n}.

Algorithm 2.2

```
Begin Master_RPCRS($f, S, N, NP, npoints, NF_{max}, \epsilon$)
  initialize:
    $Iter = 0;\ D_{max} = \epsilon + 0.1;\ ns = 0$
    Select at random a set, $R$, of $N$ trial points $R^i$; $(0 \leq i < N)$
  SEND $N/NP$ trial points from $R$ to each worker processor
  RECEIVE $N$ function values from the $NP$ processors
  do $j = 1 : NP$
    do $k = 1 : npoints$                      # $b = NP \times npoints$
      $Iter = Iter + 1$
      $\overline{G}^k = \sum_{i=0}^{n-1} R^{j_i}/n$
      $\overline{P}^k = 2 \times \overline{G^k} - R^{j_n}$.                # Primary points
    SEND $\overline{P}^k$ to processor $j$; $(k = 1, \dots, npoints)$.
  while $(D_{max} > \epsilon$ OR $f(R^W) - f(R^B) < \delta$ OR $Iter < NF_{max})$
    update: Determine the trial points $W$ and $B$ such that
          $f(R^W) \geq f(R^i) \geq f(R^B)$ $(0 \leq i < N)$
    $Iter = Iter + 1$
    $\overline{P}_{new}$ = gen_trial()
    RECEIVE $f(\overline{P})$ from processor $idp$          # $(1 \leq idp \leq NP)$
    SEND $\overline{P}_{new}$ to processor $idp$
    if $f(\overline{P}) < f(R^W)$
      $R^W = \overline{P};\ ns = ns + 1$
  End while
  Return $\{R^B$ and $f(R^B)\}$;                  # $f(R^B) \leq f(R^i)$; $(0 \leq i < N)$
End Master_RPCRS
```

In the convergence loop the greatest value of the function $(f(R^W))$ in the set $R^0, \dots, R^{N-1}$ is determined. Also a new $\overline{P}$ (named $\overline{P}_{new}$) is computed in **gen_trial()** procedure. In this procedure the algorithm will generate a primary or a secondary trial point following the same strategy of CRS. Then, master processor waits for the arrival of a new value of the objective function from any worker processor and immediately sends a new trial point $\overline{P}_{new}$ to this worker processor. After that, master processor checks if the received trial point is accepted or not and decides if the next trial point should be a primary or a secondary trial point.

Worker_RPCRS process consists of an initial stage where worker processor receives N/NP trial points from master processor, evaluates them and returns the values of the function back to master processor. Then, worker processor receives *npoints* to be evaluated. They are stored in a FIFO buffer, A. When a worker processor has evaluated a trial point it sends the value of the function back to the master processor and checks for the arrival of a new trial point. If a new point

Algorithm 2.3

```
Begin Worker_RPCRS($f, S, N, NP, npoints$)
  RECEIVE $N/NP$ trial points and store in a FIFO $A$
  Compute $f(A^i)$; $1 \leq i \leq N/NP$
  SEND $N/NP$ values of the function $(f(A^i))$ to the master processor
  RECEIVE npoints from the master and store them into $A^{first}, \ldots, A^{last}$
  while (Master_RPCRS is working)
     while there are stored points to evaluate ($A \neq \emptyset$)
        Evaluate $f(A^{first})$
        SEND $f(A^{first})$ to master processor
        if a new point has arrived from master
            RECEIVE $A^{last}$
     end while
     wait for a new point from master processor
     RECEIVE $A^{last}$
  End while
End Worker_RPCRS
```

has arrived, worker processor reads the point and pushes it into the FIFO buffer. Otherwise, worker processor goes on evaluating the next point of its buffer. If the FIFO buffer $A = \emptyset$ and master processor is still working, worker processor has to wait for a new trial point from the master processor.

Using this strategy, idle time of worker processors is reduced (even eliminated), and in addition communication overhead is decreased because communications and computations are overlapped.

3 Evaluation of RPCRS on a uniprocessor environment

In order to hide the set of problems associated to parallel implementations, such as communication overhead or bottlenecks due to intensive communications, a machine independent evaluation of RPCRS has been realized; i.e. in this section executions of RPCRS were carried out on a uniprocessor system and performance was measured versus the number of function evaluations computed during the execution of RPCRS. The goal of this analysis consists of determining the behavior of RPCRS with respect to CRS as a function of the buffer size (b).

A set of twenty two test functions has been used to check convergence and parallel performance of RPCRS. Due to the strong stochastic component of this algorithm, the number of function evaluations carried out by RPCRS depends on the particular execution. For this reason, the algorithm has been executed 100 times for every value of the setting parameters, obtaining a significantly statistic sample. From this data set, average value of the number of function evaluations and the corresponding confidence intervals (95%) were computed (see [13]). Setting lower and upper limits to a statistic implies that the probability of an interval covering the mean is 0.95 or, expressed in another way, that on the average 95 out of 100 confidence intervals similarly obtained would cover the mean.

Table 1: Results of RPCRS for Goldstein/Price and Shekel10 test functions.(1) and (2) mean 98% and 97% of success were obtained respectively.

	Goldstein/Price Test Function					
	Cluster Size = 25 $\times$ n		Cluster Size = 50 $\times$ n		Cluster Size = 100 $\times$ n	
b	NoFE	Conf.Int.	NoFE	Conf.Int.	NoFE	Conf.Int.
1	(1)1622	[1604,1640]	3254	[3230,3278]	6505	[6467,6544]
2	1650	[1632,1667]	3261	[3238,3284]	6511	[6479,6543]
3	1633	[1609,1657]	3261	[3238,3284]	6502	[6473,6532]
4	1649	[1630,1669]	3256	[3229,3283]	6501	[6464,6539]
8	1631	[1612,1649]	3235	[3207,3264]	6520	[6483,6558]
16	1599	[1575,1623]	3212	[3186,3237]	6431	[6394,6468]
32	1606	[1574,1637]	3146	[3112,3180]	6340	[6301,6378]
64	1706	[1674,1739]	3149	[3112,3186]	6229	[6195,6264]
	Shekel10 Test Function					
1	5310	[5262,5357]	10614	[10562,10666]	21454	[21367,21540]
2	(2)5315	[5260,5370]	10645	[10592,10698]	21543	[21475,21611]
3	5326	[5283,5369]	10620	[10562,10678]	21523	[21433,21613]
4	5335	[5295,5375]	10576	[10506,10646]	21498	[21425,21571]
8	5279	[5229,5329]	10575	[10509,10642]	21472	[21390,21553]
16	5161	[5117,5206]	10477	[10409,10545]	21452	[21371,21532]
32	5049	[4945,5152]	10237	[10171,10303]	21066	[20987,21145]
64	5059	[4905,5213]	9784	[9728,9839]	20660	[20577,20743]

In order to facilitate analysis of the behavior of RPCRS, in a first set of experimental tests, only six test functions were used: Goldstein/Price, Hartman3, Hartman6, Shekel5, Shekel7 and Shekel10 [15].

Two of the RPCRS's input parameters which play an important role in the performance evaluation are: the number of trial points N defining the cluster of trial points and the size b of the buffer. During the first set of tests, performance evaluation has been made as a function of both N and b. N has been established as a function of the dimension of the problem n, so values for N in our performance evaluation were $N = 25 \times n$, $50 \times n$, $75 \times n$ and $100 \times n$. Values for the buffer size b were 1, 2, 3, 4, 8, 16, 32, 64. It must be pointed out that for $b = 1$, RPCRS algorithm matches to the original sequential version of CRS algorithm. So, performance evaluation will be realized by comparing RPCRS(b) to RPCRS($b = 1$). For all the executions of RPCRS, the parameters used in the stopping criteria were: $\epsilon = 10^{-5}$, $\delta = 10^{-5}$ and $NF_{max} = 10^6$.

In Table 1 numerical results obtained from the evaluation of two test functions, using several values of b and N, are given. In this table average values of the number of function evaluations (NoFE) and the corresponding confidence intervals (Conf.Int.) for a sample of hundred executions of RPCRS, are shown.

From Table 1, it must be noticed that for the smallest value of the cluster size ($25 \times n$) the percentage of success of RPCRS were not always 100% (see notes (1) and (2)). Therefore, bigger values of the cluster size N must be chosen to ensure the convergence of the algorithm. It can be seen in Table 1 that the number of function evaluations tends to decrease when the cluster size N grows, though this tendency can not be observed for Goldstein/Price Function with the smallest cluster size value ($25 \times n$)

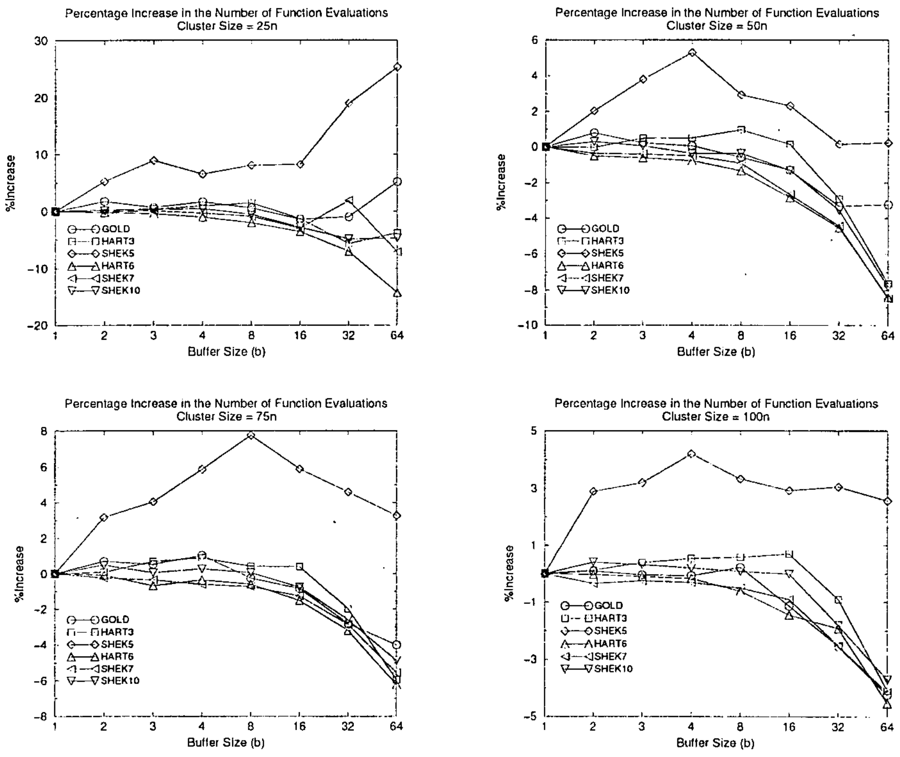

Figure 1: Percentage of increase in the number of function evaluations ($NoFE$) for several values of the cluster size (N): % Increase = $\frac{NEval(b)-NEval(b=1)}{NEval(1)} \times 100$.

Figure 1 shows results of RPCRS over the set of six test functions for the four values of cluster size. Due to the average values of the number of function evaluations range between 1000 and 40000, we decided not to draw the number of function evaluation versus the buffer size b. Instead of this, in Figure 1, the percentage of increase (%Increase) in the number of function evaluations with respect to the original case ($b = 1$), $(\frac{NEval(b)-NEval(b=1)}{NEval(1)} \times 100)$, has been drawn.

So Figure 1 shows the percentage of increase (or decrease) for several different values of the cluster size. For Shekel 5 test function, a positive %Increase were always obtained, though this increment in the number of function evaluations diminishes as the cluster size increases. For the remaining test functions, it can be seen that %Increase varies just a little bit because the average values of the number of function evaluations remains almost constant, with respect to b, when the buffer size is small ($1 \leq b \leq 8$). For bigger buffer sizes the %Increase tends to be more negative (e.g. Hartman6). Only for Goldstein/Price test function with a cluster size $N = 25 \times n$, the %Increase is positive for a buffer size $b = 64$. This is due to the number of points stored in the buffer is relatively large compared with the cluster size. Goldstein/Price test function is two-dimensional and therefore the cluster size in this case is smaller, $25 \times n = 50$, than the buffer size ($b = 64$); i.e. $\frac{N}{b} < 1$.

Results from this set of experiments seems to show that: (i) using a cluster size large enough, convergence to the global optimum is ensured and (ii) for a value of the buffer size smaller than the cluster size but greater than 8 the computational cost of RPCRS diminishes or remains similar to the original CRS (RPCRS($b = 1$)).

In a second set of experiments, performance evaluation of RPCRS was made for a wider set of test functions (see appendix for a detail description of these test functions). In this case the cluster size was always $N = 100 \times n$, ensuring that for all the test functions $N > b$. This new set of test functions includes all the functions previously used and seventeen additional functions. These functions have been chosen in such a way that they are defined over several different domains of definition, $S \subset R^n$, where S ranges from $[-1,1]$ to $[-600,600]$ and n from 1 to 10; the number of global optima varies from 1 to > 10 and at least one of the functions has more than 1000 local optima. For all the functions, 100% of success in finding the optimal solution was obtained by RPCRS.

In Figure 2, performance evaluation of RPCRS, for the set of 22 test functions, is shown. In this graph, X-axe represents the index of the test function. The functions have been sorted by the increasing number of function evaluations needed to reach the global solution when $b = 1$ ($NEval(b = 1)$. For each function, results for all the values of the buffer size (b) are displayed in a vertical straight line. Roughly speaking it can be said that when the number of function evaluations is big enough the performance of RPCRS is best than that of CRS; i.e. $NEval(b) < NEval(b = 1)$ for most of the values of b. For test functions with a computational burden not too strong, the performance of the algorithm depending of the value of the buffer size, has more fluctuations. Anyway it can be seen that only for 5 functions over the set of 22 test functions, RPCRS performs worst than CRS; i.e $NEval(b) > NEval(b = 1)$.

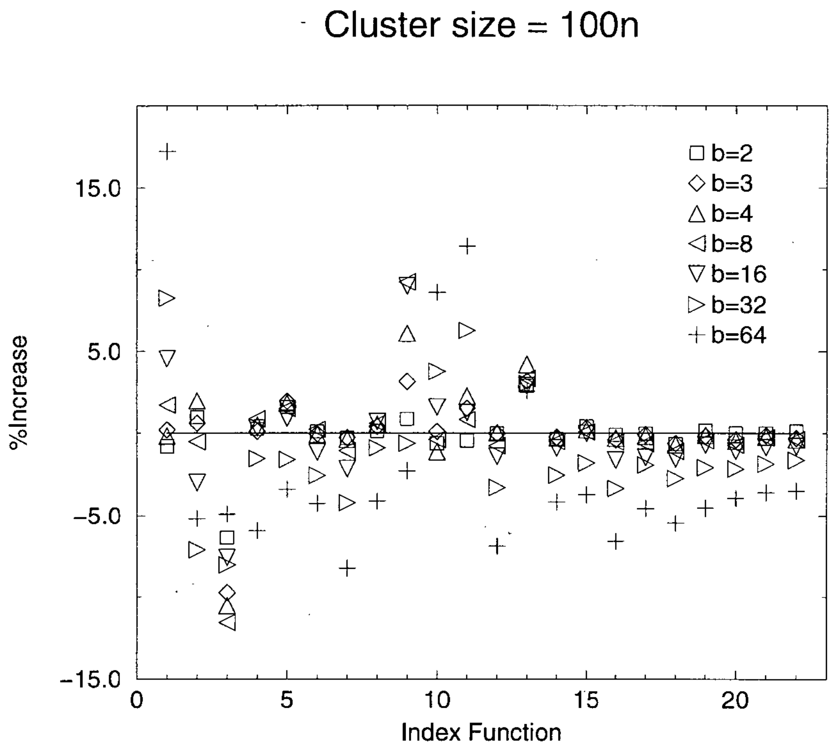

Figure 2: Percentage of increase in the number of function evaluations for several values of the buffer size (b) and $N = 100 \times n$. % Increase $= \frac{NEval(b)-NEval(b=1)}{NEval(1)} \times 100$.

In general, it can be said that the parallelism introduced at CRS for building RPCRS does not strongly disturb its performance characteristic and for hard functions, requiring a lot of computational resources, RPCRS outperforms CRS. Nevertheless, no theoretical proof of these results has been still studied. In the next section, results of the performance evaluation of RPCRS on a parallel system (Cray T3D) using up to 16 processors are presented.

4 Performance evaluations on a parallel system

In order to analyze the behavior and the performance of the parallel program, we have chosen a cluster size of $100 \times n$. In our experiments buffer sizes $b = 16$ and $b = 64$ were used. Though our asynchronous parallel program was designed with the capability of overlapping computations and communications, the algorithm needs

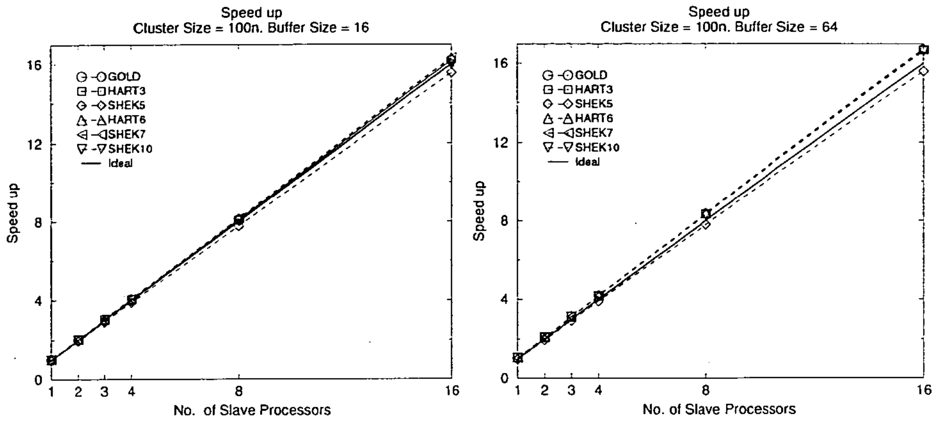

Figure 3: Speedup of the parallel executions of RPCRS with respect to the sequential case. Speed-up $=t(RPCRS(b=1,p=1))/t(RPCRS(b,p))$. Delay = 0.03 sec.

the computational cost of the test functions to be greater than the communication time required to exchange data among master and worker processors. If computational cost of the objective function is small enough, bottlenecks would appear at master processor. But for these unexpensive functions it would not be necessary the use of a parallel system. So, in our evaluations of the parallel performance of RPCRS, it was simulated that test functions have the same computational cost by introducing an additional time delay into the function evaluation. Particularly, effects of delays: 0.003 sec. and 0.03 sec. have been analyzed for a set of six test functions.

Figure 3 shows the values of the speedup obtained when the execution time for the parallel executions is compared to the execution time obtained by the original sequential algorithm ($CRS = RPCRS(b=1,p=1)$); i.e. Speed-up $=t(RPCRS(b=1,p=1)/t(RPCRS(b,p))$, where p is the number of worker processors. From results at Figure 3, it might seem that we have implemented a marvelous parallel program because of a speedup over linear is most of times obtained. Nevertheless, these super speedups are due to the property that the number of function evaluations carried out by RPCRS algorithm is lesser for buffer sizes $b=16$ or $b=64$ than for a buffer size $b=1$ (see Figure 1).

Figure 4 shows values of the speedup obtained when execution time for the parallel program is compared to execution time obtained in the sequential case, but in this case using the same value of the buffer size for both sequential and parallel executions; i.e. Speed-up $= t(RPCRS(b,p=1)/t(RPCRS(b,p))$. It can be seen that in this case an almost linear speedup has been obtained for all the functions, but no super speedups. These speedups are closer to the linear speedup

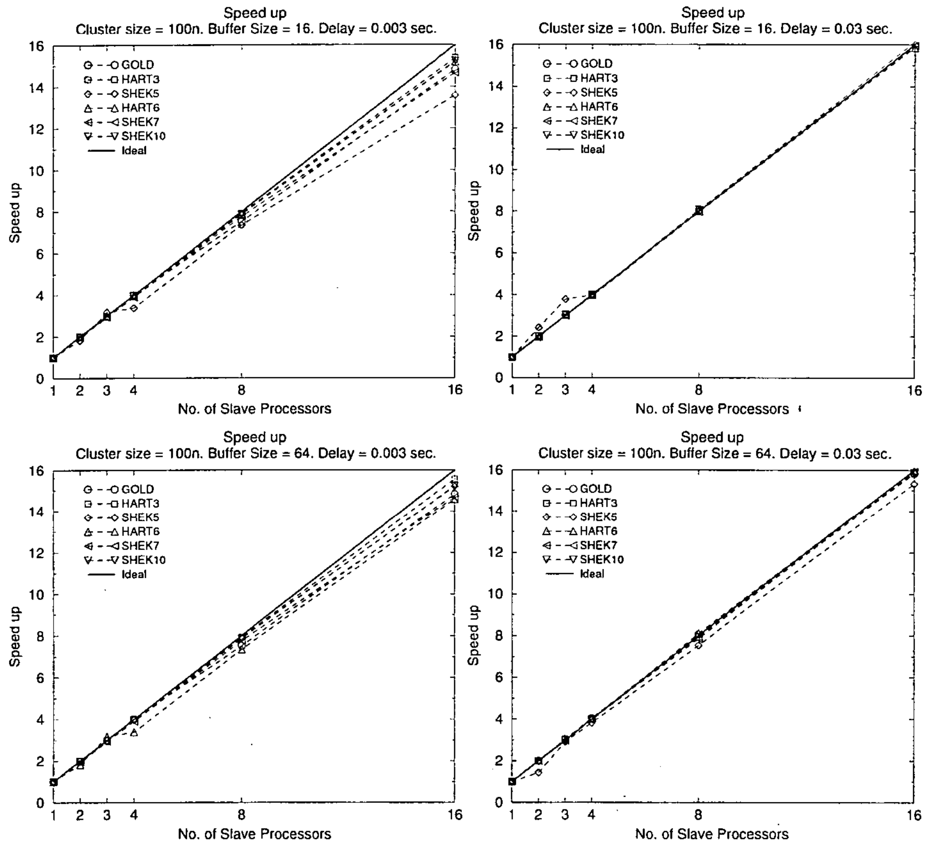

Figure 4: Speedup of the parallel version with respect to the sequential case. Speed-up = $t(RPCRS(b, p = 1))/t(RPCRS(b, p))$.

for a delay of 0.03 sec. than for a delay of 0.003 sec.

5 Conclusions and future work

In this work a parallel implementation of CRS, called RPCRS, has been described and evaluated. It has been shown that RPCRS is computationally cheaper than CRS and in addition it is easy to implement on a real parallel or distributed computing system an asynchronous model.

Although a wide set of standard test functions were used to validate that RPCRS outperforms to CRS, no theoretical support exists behind this work to demonstrate that our results will be the same for any function. Our future work will be aimed at obtaining a better understood of this results and a mathematical proof (if any).

Appendix: Description of the test functions

F :	Index of the Function.
D :	Search domain.
$f(x^*)$:	Global minimum value of the function.
M :	Number of global plus local minima of the function.

Table 2: Description of the test functions.

F	Function f(x)	D	$f(x^*)$	M
1	Three hump camel back	$[-5,5]^2$	0.0	3
2	$(x_1-5)^2-(x_2-10)^2$ if $x_1 \leq 10$ $(x_1-15)^2-(x_2-10)^2$ otherwise	$[0,20]^2$	0.0	2
3	Six hump camel back	$[-2.5,2.5]^2$	-1.0316	6
4	Booth	$[-5,5]^2$	0.0	1
5	Levy 13	$[-10,10]^2$	0.0	≥ 1
6	Goldstein / Price	$[-2,2]^2$	3.0	3
7	Shperical 2 $= \sqrt{\sum_i^2 x_i^2}$	$[-1,1]^2$	0.0	1
8	Hartman 3	$[0,1]^3$	-3.862782	≥ 3
9	Beale	$[-5,5]^2$	0.0	> 4
10	Levy 3	$[-10,10]^2$	-176.54	≥ 9
11	Griewank	$[-600,600]^2$	0.0	≥ 10
12	Shperical 3 $= \sqrt{\sum_i^3 x_i^2}$	$[-1,1]^3$	0.0	1
13	Shekel 5	$[-10,10]^4$	-10.15320	≥ 4
14	Shekel 7	$[-10,10]^4$	-10.40294	≥ 4
15	Shekel 10	$[-10,10]^4$	-10.53641	≥ 4
16	Shperical 4 $= \sqrt{\sum_i^4 x_i^2}$	$[-1,1]^4$	0.0	1
17	Hartman 6	$[0,1]^6$	-3.322828	≥ 6
18	Shperical 5 $= \sqrt{\sum_i^5 x_i^2}$	$[-1,1]^5$	0.0	1
19	Shperical 6 $= \sqrt{\sum_i^6 x_i^2}$	$[-1,1]^6$	0.0	1
20	Shperical 7 $= \sqrt{\sum_i^7 x_i^2}$	$[-1,1]^7$	0.0	1
21	Shperical 8 $= \sqrt{\sum_i^8 x_i^2}$	$[-1,1]^8$	0.0	1
22	Shperical 9 $= \sqrt{\sum_i^9 x_i^2}$	$[-1,1]^9$	0.0	1

References

[1] Duckbury, P.G., *Parallel Array Processing.* Chichester: Ellis Horward, 1986.

[2] García, I., Ortigosa, P.M., Casado, L.G., Herman, G.T. and Matej, S., "A parallel implementation of the controlled random search algorithm to optimize an algorithm for reconstruction from projections," in *IIIrd Workshop on Global Optimization,* (Szeged, Hungary), pp. 28–32, December 1995.

[3] García, I. and Herman, G.T., "Global optimization by parallel constrained biased random search," in *State of Art in Global Optimization: Computational Methods and Applications* (C.A. Floudas and P.M. Pardalos, ed.), Kluwer Inc, pp.433-455, 1996.

[4] García, I., Ortigosa, P.M., Casado, L.G., Herman, G.T. and Matej, S., "Multidimensional Optimization in Image Reconstruction from Projections," in *Developments in Global Optimization,* (L.M. Bomze, T. Csendes, R. Horst and P.M. Pardalos eds,), Kluwer Inc, pp. 289–300, 1997.

[5] Hendrix, E.M.T., *Global Optimization at Work,* PhD. Dissertation, Wageningen Agricultural University, 1998.

[6] Horst, R. and Pardalos, P.M. eds., *Handbook of Global Optimization,* Dordrecht: Kluwer, 1995.

[7] McKeown, J.J., "Aspects of parallel computations in numerical optimization," in *Numerical Techniques for Stochastic Systems* (F. Arcetti and M. Cugiani, eds.), pp. 297–327, 1980.

[8] Nelder, J.A. and Mead, R., "A simplex method for function minimization," in *The Computer Journal,* pp. 308–313, 1965.

[9] Price, W.L., "A controlled random search procedure for global optimization," in *Towards Global Optimization 2* (L.C.W Dixon and G.P. Szegö, eds.), pp. 71–84, Amsterdam: North Holland, 1978.

[10] Price, W.L., "A controlled random search procedure for global optimization," in *The Computer Journal,* no. 20, pp. 367-370, 1979.

[11] Price, W.L., "Global optimization algorithms by controlled random search," *Journal of Optimization Theory and Applications,* no. 40, pp. 333–348, 1983.

[12] Price, W.L., "Global optimization algorithms for a CAD workstation," *Journal of Optimization Theory and Applications,* no. 55, pp. 133–146, 1987.

[13] Sokal, R.R. and Rohlf, F.J., *Biometry.* New York: W. H. Freeman and company, 1981.

[14] Sutti, C., "Local and global optimization by parallel algorithms for MIMD systems," *Annals of Operating Research,* no. 1, pp. 151–164, 1984.

[15] Törn, A. and Zilinskas, A., *Global Optimization. Lecture Notes in Computer Science 350.* Springer-Verlag, Berlin, 1989.

[16] Woodhams, F.W.D. and Price, W.L., "Optimizing accelerator for CAD workstation," *IEE Proceedings Part E*, vol. 135, no. 4, pp. 214–221, 1988.