Mathematical Programming 46 (1990) 1-29
North-Holland

CONCURRENT STOCHASTIC METHODS FOR GLOBAL OPTIMIZATION

Richard H. BYRD

Department of Computer Science, University of Colorado, Boulder, Colorado, 80309, USA

Cornelius L. DERT

Econometric Institute, Erasmus University Rotterdam, The Netherlands and
Department of Computer Science, University of Colorado, Boulder, Colorado, 80309, USA

Alexander H.G. RINNOOY KAN

Econometric Institute, Erasmus University Rotterdam, The Netherlands

Robert B. SCHNABEL

Department of Computer Science, University of Colorado, Boulder, Colorado, 80309, USA

Received 17 November 1986
Revised manuscript received 10 May 1988

The global optimization problem, finding the lowest minimizer of a nonlinear function of several variables that has multiple local minimizers, appears well suited to concurrent computation. This paper presents a new parallel algorithm for the global optimization problem. The algorithm is a stochastic method related to the multi-level single-linkage methods of Rinnooy Kan and Timmer for sequential computers. Concurrency is achieved by partitioning the work of each of the three main parts of the algorithm, sampling, local minimization start point selection, and multiple local minimizations, among the processors. This parallelism is of a coarse grain type and is especially well suited to a local memory multiprocessing environment. The paper presents test results of a distributed implementation of this algorithm on a local area network of computer workstations. It also summarizes the theoretical properties of the algorithm.

Key words: Global optimization, concurrent, parallel, stochastic, network of computers.

1. Introduction

This paper presents a parallel algorithm for the global optimization problem. The algorithm is a stochastic method related to the multi-level single-linkage method of Rinnooy Kan and Timmer (1984) for sequential computers. The parallelism is of a coarse grain type and is especially well suited to a local memory multiprocessing environment. The paper presents test results of a distributed implementation of this algorithm on a local area network of computer workstations. It also summarizes the theoretical properties of the algorithm.

Research supported by AFOSR grant AFOSR-85-0251, ARO contract DAAG 29-84-K-0140, NSF grant DCR-8403483, and NSF cooperative agreement DCR-8420944.

The global optimization problem is to find the lowest function value of a function that may have multiple local minimizers. We denote the problem as follows:

$$\begin{aligned} &\text{given} \quad f(x): R^n \to R \quad \text{and} \quad S \subseteq R^n \\ &\text{find} \quad x_* \in S \ \text{for which} \ f(x_*) \leq f(x) \ \text{for all} \ x \in S. \end{aligned} \tag{1.1}$$

We refer to x_* as the *global minimizer* of f and $f(x_*)$ as the *global minimum*. The term "global" contrasts with a *local minimizer* of f, which is the lowest value of $f(x)$ in some open neighborhood in S. Thus a function may have multiple local minima but it can have only one global minimum value. In this paper we assume that f is a nonlinear, twice continuously differentiable function. We also assume that the feasible region S is given by a set of lower and upper bounds on each variable, i.e.

$$S = \{x \mid l_i \leq x_i \leq u_i, \quad i = 1, \ldots, n\}$$

and that the global minimizer lies in the interior of S.

Global optimization problems of the above form occur in many practical applications including data fitting, structural design, optimal control, econometrics, and many more (see e.g. Dixon and Szego (1978)). Most optimization software, however, is only constructed to find local minimizers (see e.g. Gill, Murray and Wright (1981) or Dennis and Schnabel (1983)). One reason is that it is difficult to construct reliable algorithms for finding the global minimizer of general nonlinear functions. A second reason is that it often is very expensive to solve the global optimization problem reliably, since it requires many evaluations of $f(x)$ and many iterations and arithmetic operations within the optimization code itself. In practical optimization applications, the evaluation of $f(x)$ is often very expensive so that the large number of function evaluations is the dominating expense.

There has been a moderate amount of work done in developing global optimization algorithms for sequential computers. Among these approaches, we shall favor *stochastic* methods, which include some random sampling of the function domain, because these methods provide some guarantee of their reliability while still appearing at least as efficient as other approaches.

Due to the expensive nature of global optimization and the practical need to solve such problems, there is ample incentive to devise parallel global optimization methods if they can lead to significantly faster solution of this problem. It appears that the global optimization problem is well suited to parallel solution in a number of ways.

In this paper we concentrate on the high level, coarse grain concurrency available within the global optimization algorithm itself. Obvious opportunities for high level parallelism include conducting multiple local minimizations concurrently, or evaluating the objective function f at multiple random sample points concurrently. Such an approach is intended to utilize multiple processors efficiently in solving the global optimization problem whether or not the evaluation of $f(x)$ is expensive.

An alternative, lower level use of parallelism in global optimization would be to apply a parallel algorithm to evaluate $f(x)$. In cases where the function evaluation is very expensive this approach might be very effective. We do not pursue it for several reasons. First, parallelization of objective function evaluation is strongly problem dependent and outside the realm of the optimization algorithm designer. Second, if one has an efficient parallel code for evaluating $f(x)$, then it can be used in conjunction with the approach presented here. For example, if evaluation of $f(x)$ vectorizes well (as is often the case), then the appropriate computational environment would be a multiprocessor where each node is a vector processor. In this environment the function evaluations could be performed on individual vector processors while the higher level parallelism in our algorithm could still be realized among these processors. Indeed, we believe that this computational environment (currently embodied by machines including the Cray X-MP and the Alliant) will be a very important one in the future. In other cases where the computation of $f(x)$ requires multiple processors, it would be possible to divide the processors into groups, with a group of processors being used to evaluate $f(x)$ and the groups themselves being used to implement the high level parallelism of our algorithm.

The implementation of the type of coarse grain parallel algorithm we develop requires a computer capable of executing multiple independent instruction streams at the same time. In the taxonomy of Flynn these are known as *Multiple Instruction Multiple Data* (MIMD) computers. The class of MIMD computers includes both shared and local memory multiprocessors.

Shared memory multiprocessors are computers with multiple processors that are all connected, via a switching network or global bus, to a shared memory which they all can access. The processors may have local memory as well. Synchronization or communication between processors is carried out using this shared memory and usually is nearly as fast as a local memory access.

Local memory multiprocessors are computers with multiple processors, each with its own local memory, connected by some sort of interconnection network. Examples currently in use include the hypercube computers pioneered at Caltech (Seitz (1985)) and local area networks of computers. On these machines, synchronization or communication between processors is achieved by passing messages between the processors. Generally message throughput rates are several orders of magnitude slower than the arithmetic operation rate. Thus local memory multiprocessors are best suited to parallel algorithms where, on the average, many instructions (say 1000 or more) are executed on individual processors in between synchronization or communication points with other processors. Such algorithms are generally referred to as *medium grain* or *coarse grain* parallel algorithms.

Since the global optimization problem is amenable to solution by a coarse grain parallel algorithm, a local memory multiprocessor is an appropriate parallel environment for this problem. For this reason, we have chosen to implement our parallel algorithm on the local area network of workstations that is being used for parallel computation at the University of Colorado. Due to the small amount of synchronization and communication in our parallel algorithm, our experimental results are likely to be indicative of the performance we would obtain in other MIMD environments.

In summary, the goal of this research was to develop an efficient and reliable parallel algorithm for the global optimization problem that is well suited to implementation in a local memory multiprocessing environment. By reliable, we mean that the algorithm should be successful in finding the global minimum and preferably that there be some theoretical guarantee of this reliability. By efficient, ideally we mean that our parallel algorithm, when implemented on P identical processors, should require $1/P$ of the time that the best sequential algorithm would on one of these processors to solve the same problem.

To approach this efficiency goal, three important performance goals must be met. First, the algorithm should keep all processors (nearly) fully busy, i.e. processor idle time should be minimized. Second, the algorithm should introduce little new work that was not required by the best sequential algorithm; this rules out extensive overhead computations introduced in order to utilize multiple processors. Third, the interprocess communication requirements, in this case the number of messages, should be small. The experimental results in this paper will show that these goals have been met quite well.

In Section 2 we briefly describe sequential methods for global optimization, concentrating on the approach from which our parallel algorithm is derived. Section 3 presents our parallel global optimization method. In Section 4 we briefly summarize the theoretical properties of this method. In Section 5 we first describe the multiprocessing environment, a network of computer workstations, used in our experiments. Then we present our computational results in this environment. Some comments on future directions for this work are presented in Section 6.

2. Sequential global optimization methods

The methods that have been developed to solve the global optimization problem can be divided into two main classes, *deterministic* methods and *stochastic* methods (Dixon and Szego (1978)). This section briefly surveys these methods, concentrating on those most closely related to our concurrent algorithms.

Deterministic methods do not incorporate any random or stochastic features. A wide variety of approaches are contained in this class, including trajectory methods (Branin (1972), Branin and Hoo (1972)), deflation methods (Goldstein and Price (1971), Levy and Gomez (1985), Levy and Montalvo (1985)), piecewise approximation methods (Shubert (1972)), and interval arithmetic methods (Hansen (1980), Hansen and Sengupta (1980), Walster, Hansen and Sengupta (1985)). Most of these methods either do not provide a guarantee that they will find the global minimizer, or do so only at the expense of making additional assumptions about the objective function f which are difficult to verify in practice. The most common such assumption is that some derivative of f obeys a Lipschitz condition with a constant that is bounded above by a known number. In addition, many deterministic methods tend to require a large computational effort to find the global minimizer.

Stochastic methods differ in that they incorporate stochastic features, generally the sampling of $f(x)$ at randomly selected points in the feasible region. This enables these methods to provide a probabilistic guarantee that the global minimizer will be found, assuming only that f is continuously differentiable. Generally, these methods combine the random sampling phase with a phase where local minimization algorithms are performed from some of the sample points. The earliest stochastic methods, such as random search (Brooks (1958), Anderssen (1972)), random direction (Devroye (1979), Price (1979), Solis and Wets (1981)) and simple multi-start methods, were rather crude and not computationally efficient. More recent stochastic methods, such as those of Boender, Rinnooy Kan, Stougie and Timmer (1982), and Rinnooy Kan and Timmer (1984, 1985a,b,c), combine random search and local minimization carefully and appear to be quite efficient in computational experiments. In addition, they also provide probabilistic guarantees of their computational efficiency.

Thus, modern stochastic methods appear to provide an attractive choice from both the theoretical and computational points of view. For these reasons, we have chosen to base our concurrent global optimization algorithms on a stochastic approach.

In particular, our concurrent method is most closely related to the recent *multi-level single linkage* method of Rinnooy Kan and Timmer (1984). This method appears to combine state-of-the-art computational performance with strong theoretical properties. The remainder of this section briefly reviews sequential multi-level single linkage methods, with emphasis on aspects that will have importance for our concurrent methods.

The multi-level single linkage method is an iterative algorithm. Each iteration consists of a sampling phase, in which the function is evaluated at a number of randomly sampled points, followed by a minimization phase, in which a local minimization procedure is started from a subset of the sample points. A probabilistic stopping rule is applied to determine whether the algorithm should be continued, and if it should, the next iteration is begun. Here, an outline of the algorithm is given.

Algorithm 2.1. *Multi-level single linkage method for global optimization*

Given $f: R^n \to R$, feasible region S.

At iteration number k:

1. *Generate sample points and function values.* Add N points, drawn from a uniform distribution over S, to the (initially empty) set of sample points, and evaluate $f(x)$ at each new sample point.
2. *Select start points for local searches.* (Optional: calculate a cut off level; all sample points with function values above this level will be excluded from the start point selection.)

Determine a (possibly empty) subset of the sample points from which to start local searches.

3. *Perform local minimizations from all start points.*
4. *Decide whether to stop.* If stopping rule is satisfied, regard the lowest local minimizer found as the global minimizer, otherwise go to Step 1.

Several portions of Algorithm 2.1 require further elaboration. Most important to the practical and theoretical success of the method is the selection of start points for local minimizations in Step 2. At iteration k, each sample point x is selected as a start point for a local minimization if it has not been used as a start point at a previous iteration, and if there is no sample point y within the *critical distance* $r(k)$ of x with a lower function value, i.e. with

$$\|x-y\| \leq r(k) \quad \text{and} \quad f(x) < f(y).$$

The critical distance is given by

$$r(k) = \pi^{-1/2}\left[\Gamma\left(1+\frac{n}{2}\right) m(S)\sigma \frac{\log kN}{kN}\right]^{1/n} \tag{2.1}$$

where $m(S)$ denotes the Lebesque measure of S, Γ denotes the gamma function, σ is a positive constant, and N is the sample size per iteration.

The above selection procedure may optionally be applied only to the γkN sample points with the lowest function values, where γ is any fixed number in $(0, 1]$. This corresponds to the application of a *cut off level* to the sample. The use of a cut off level does not affect the theoretical reliability and efficiency of the multi-level single linkage method, but appears to enhance its computational performance.

The local minimizations are performed by any standard unconstrained minimization code. The theoretical analysis of the multi-level single linkage method simply assumes that the unconstrained minimization code will find a local minimizer x^* when started within the basin of x^*, i.e., the set of points x from which all strictly descent paths converge only to x^*. The methods of Rinnooy Kan and Timmer (1985a,c) use the VA10AD variable metric subroutine from the Harwell Subroutine Library, while the methods reported in this paper use the line-search BFGS code in the UNCMIN package of Schnabel, Koontz and Weiss (1985). Both are well-tested and widely used codes.

A Bayesian stopping rule of Boender and Rinnooy Kan (1984) (see also (Zielinski (1981))) is applied in Step 4. To explain this rule, let w denote the number of local minimizers found after k iterations, and let $s = \gamma kN$ be the (reduced) sample size after k iterations. In addition, define the region of attraction of a local minimizer x^* for a particular local search method to be the set of all starting points x from which the local search method will converge to x^*. Boender and Rinnooy Kan show that a Bayesian estimate of the total number of local minimizers is given by

$$\frac{w(s-1)}{s-w-2} \tag{2.2}$$

and that a Bayesian estimate of the portion of S covered by the regions of attraction of the local minimizers found so far is given by

$$\frac{(s-w-1)(s+w)}{s(s-1)}. \tag{2.3}$$

The stopping rule used is that the algorithm is terminated after the kth iteration if and only if the estimate given by (2.2) is greater than w by less than 0.5, and the estimate given by (2.3) is $\geqslant 0.995$.

Strong theoretical properties of the multi-level single linkage algorithm have been proven in Timmer (1984), Rinnooy Kan and Timmer (1985b,c). If the critical distance is given by (2.1) with $\sigma > 0$, then with probability 1, all the isolated local minimizers of $f(x)$ will be found within a finite number of iterations. If $\sigma > 4$ in (2.1), then, even if the sampling continues forever, the total number of local searches started by the algorithm will be finite with probability 1. Thus both the accuracy and the efficiency of the method are guaranteed in a strong probabilistic sense.

Test results for the multi-level linkage algorithm are reported in Rinnooy Kan and Timmer (1985a,c). The algorithm has been tested on a standard set of test problems and compared with a number of other approaches for global optimization. Overall, it seemed to offer the best combination of efficiency and reliability of the methods tested.

3. A concurrent algorithm for stochastic global optimization

The global optimization problem seems conducive to solution by highly parallel algorithms in ways that makes it suitable to a variety of parallel computing environments. In particular certain methods to solve the problem can be decomposed into a number of relatively large and independent subtasks. Concurrent algorithms can exploit this coarse grain parallelism while requiring only infrequent interprocess communication and little or no shared memory. Thus these algorithms seem well suited to a local memory or shared memory multiprocessing environment. In this section we discuss one such global optimization algorithm, a synchronous concurrent multi-level single linkage method.

We can readily identify three sources of high level parallelism in the multi-level single linkage method. In the first phase of each iteration, the sampling phase, each processor can generate $1/P$ of the sample points (P is the number of processors) and evaluate the function at each of them. In the second phase, start point selection, each processor can select start points from its own subsample. (Some checking in other subsamples may be required; this is discussed later.) Finally in the third phase, local minimization, each processor can be responsible for one or more of the local searches.

An obvious mechanism for achieving this concurrency is to divide the feasible region into P subregions of equal size, and assign each subregion to a different processor. Then the sampling phase can be implemented concurrently simply by having each processor sample its subregion, and the major part of the start point selection can be accomplished by having all the processors concurrently generate the start points for their subregions based on their own samples. It is possible, however, that the number of start points for local searches may vary widely between subregions, and the lengths of local searches also may vary widely. Thus it may not be advantageous to have each processor simply handle the local searches for the start points from its own region. Instead, in our algorithm a master process collects all the start points from all the subregions and then distributes them back to the processors as evenly as possible. This is discussed in more detail below. The master process also coordinates the small amount of synchronization and communication that is required.

At this point it may be useful to indicate a basic difference between the second source of parallelism described above and the other two sources. The first and the third phases of the algorithm, sampling and local minimization, are generally dominated by the costs of the evaluations of $f(x)$. Thus our concurrent algorithm essentially is distributing these function evaluations among the processors. In the second phase, start point selection, however, there are no function evaluations, and our concurrent algorithm is carrying out part of the global optimization algorithm itself in parallel. The impact of the latter type of concurrency relative to the former on the overall speedup of the algorithm clearly will be determined by the percentage of time spent in various phases of the algorithm; this in turn will depend on the cost per function evaluation and the number of function evaluations required to solve a particular problem. The more time spent on function evaluations, the more important the concurrency from the sampling and local minimization phases of the algorithm. In fact, if function evaluations are sufficiently expensive it may be profitable—as discussed before—also to exploit a lower level of parallelism, performing parallel function evaluations within the individual local minimizations (see Section 6).

A concurrent multi-level single linkage algorithm employing P processors and making use of all three high level sources of parallelism is outlined in the following algorithm:

Algorithm 3.1. *A concurrent multi-level single linkage method for global optimization*

Given $f: R^n \to R$, feasible region S and P processors.

0. *Partition S.* Subdivide S into P equal size, regular shaped subregions S_i, $i = 1, \ldots P$, and assign subregion S_i to processor i for $i = 1, \ldots, P$.

At iteration number k:

1. *Generate sample points and function values.* For $i = 1, \ldots, P$:
   Add N/P points, drawn from a uniform distribution over subregion i, to the (initially empty) set of sample points, and evaluate $f(x)$ at each new sample point.

2. *Select start points for local searches.* [Optional: calculate a cut off level; all sample points with function values above this level will be excluded from the start point selection.]
   For $i = 1, \dots, P$:
   Determine a (possible empty) set of start points in subregion i, disregarding sample information from all other subregions.
   Resolve start points near borders between subregions.
3. *Perform local minimizations from all start points.* Collect all start points and distribute one to each processor, which performs a minimization from that point. Issue a new start point to a processor as soon as it terminates its current local search, until local searches from all start points have been completed.
4. *Decide whether to stop.* If stopping rule is satisfied, regard the lowest local minimizer found as the global minimizer, otherwise go to Step 1.

The four basic steps of this algorithm are identical to those of the sequential multi-level single linkage algorithm, Algorithm 2.1. Concurrency is achieved in the implementation of each of these steps, with the exception of Step 4, which hardly contributes to the algorithm's running time. The remainder of this section consists of a more detailed discussion of Algorithm 3.1. In Subsections 3.1–3.3 we focus on the aspects where the concurrent algorithm differs from the sequential method in Steps 1, 2 and 3, respectively. In Subsection 3.4 we make some comments on how the concurrent algorithm is expected to meet the goals of reliability and efficient utilization of a local memory multiprocessing environment that were mentioned in the introductory section.

3.1. The generation of sample points and function values

In the sampling phase, each processor i extends its set of sample points by N/P new points and evaluates the function f at each of them. Whereas the random sampling was done from a uniform distribution over the entire region S in the sequential method, in the concurrent algorithm each processor generates a random sample from a uniform distribution over its own subregion S_i. As will be discussed in Section 4, this necessitates that the theoretical analysis of the algorithm be modified but it turns out not to alter the theoretical properties of the method.

3.2. The selection of start points for local searches

The selection of start points for local minimizations is carried out in two phases (three if the optional sample reduction procedure is applied). The first, local, phase is performed independently and concurrently by each processor. Processor i selects the points in its own subregion S_i which locally satisfy the start point selection rule described in Section 2 for the multi-level single linkage method. That is, at iteration k each sample point $x \in S_i$ is selected as a candidate start point if it has not been used as a start point in a previous iteration, and if there is no sample point $y \in S_i$ within the critical distance $r(k)$ of x given by eq. (2.1) with a smaller function value.

For a given sample over the entire region S, any sample point that would be selected as a start point by the sequential algorithm will also be selected as a candidate start point by this first start point selection phase of the concurrent algorithm. However, if x is within the critical distance of any border of its subregion, it is possible that it will be selected as a candidate start point by the first phase of the concurrent algorithm but not by the sequential algorithm, because some sample point in another subregion but within the critical distance has a smaller function value. To prevent the initiation of unnecessary local searches from these points, the local selection phase of the concurrent algorithm is followed by a second, global selection step. First, all candidate start points within the critical distance of a border between subregions are distributed to all processors. Then, each processor determines whether its sample contains a point within the critical distance of one of these candidate start points with a lower function value. If so, this candidate point is not used as a start point for a local minimization. The remaining start points will be the same ones that would have been selected from the same sample by the sequential method.

The start point selection procedure may optionally be preceded by a sample reduction procedure. As in the sequential algorithm, the aim of this step is to retain only the γkN sample points with the lowest function values in the entire region S, where $\gamma \in (0, 1)$ is fixed. In the sequential algorithm, a cut off level equal to the γkNth lowest function value is determined and all sample points with higher function values are eliminated. In order to determine a corresponding cut off level for the entire region S in the concurrent method, some exchange of information between subregions is required.

In order to keep both the interprocess communication and the computational costs of this step small, the goal of the sample reduction phase for the concurrent algorithm is relaxed slightly. We require that the cut off level be chosen so that the number of reduced sample points, i.e. all sample points with function values below the cut off level, deviates from the target size γkN by at most $\phi\gamma kN$ for a fixed precision $\phi \in (0, 1)$. This can be achieved by setting $g = \lfloor (2\theta\gamma kN + P - 1)/P \rfloor$ and requiring each subregion to determine the (ig)th largest function value $(i = 1, \ldots, \lfloor kN/(2\theta\gamma kN + P - 1) \rfloor)$ and to report those to the master process. The cut off level in each region is then taken to be equal to Gth largest function value in this selection, where G is the integer closest to $(2\gamma kN - (P-1)(g-1))/(2g)$. It is not hard to see that this procedure determines the cut off level within the relative accuracy required (Dert (1986)).

3.3. The local minimizations

The distribution of the computational effort in the local minimization phase is fairly straightforward. After all the local search start points have been identified in the previous step, all these points are reported to the master process. The master process then distributes one start point to each processor, and if there are more start points than processors, a processor that completes one local search is given another start point until they all have been processed.

As discussed in the beginning of this section, this phase may not keep all the processors equally busy. The remaining issue is in what order to assign the local minimizations in order to keep processor idle time as small as possible. If we knew in advance how long each local search would take, we could use a scheduling heuristic to minimize the total time, and thus the idle time, needed to complete all minimizations. A well known, simple heuristic with attractive theoretical properties for similar scheduling problems is the *longest processing time* rule (see e.g. Frenk and Rinnooy Kan (1986)), which states that the jobs should be scheduled in order of descending processing time.

In the case of local minimizations we do not know in advance how long each minimization will take, so we cannot order the start points according to processing time. One crude way to estimate these times is to guess that the higher the function value at the start point, the longer the minimization will take. Our computational experiments indicate that the use of this heuristic has given slightly better results than using an arbitrary ordering and at least as good results as any other heuristic we have attempted.

3.4. Comments

In this subsection we make some general remarks concerning the expected efficiency and reliability of the concurrent global optimization algorithm that we have just described. We will also examine how effectively the algorithm is likely to utilize a local memory multiprocessing environment.

From the point of view of the sample points that are used, the local minimizations that are performed, and the answer that is found, the concurrent algorithm differs from the sequential one only in that it samples from a slightly different distribution. Therefore we expect that the number of sample points, function evaluations and local searches that are used by the sequential and concurrent methods on any particular problem will be very similar. Furthermore we expect that the number of minimizers found by the two methods will be roughly the same, and that they usually will find the same global minimizer. This similarity between the two algorithms is reinforced by the theoretical analysis discussed in Section 4.

Let us now turn to the parallel characteristics of Algorithm 3.1. Notice that it requires very little synchronization of processes or communication or sharing of information between them. Information is exchanged at four places in the algorithm: after the local phase of Step 2 is completed, the candidate start points within the critical distance of the subregion border must be collected and sent to the other subregions; after the border resolution phase of Step 2 is completed, the final start points must be collected and distributed to the processors; after the local minimizations are completed in Step 3 each process must report the minimizers found to the process making the stopping test; and at the beginning of the next iteration, some results from the previous iteration must be distributed to the processors. The only synchronization requirements are inherent in these actions: the local phase of Step 2 must be completed by all subregions before the global phase is begun, and all the local searches must be completed before the stopping test is made. If sample reduction is used, the implementation of Algorithm 3.2 also requires that every gth function value from each subregion be collected from all processors and that the cut off level be communicated back to each processor.

In our implementation, a master process, which resides on the same processor as one of the subregion processes, takes care of the coordinating activities described above. It collects the candidate start points that are near subregion borders and distributes them to the subregion processes when it has all of them; it collects the start points and distributes them to the processors using the heuristic discussed in Section 3.3; and when all local searches are completed, it performs the stopping test (two simple equations) and starts the next iteration if required. If sample reduction is used it also collects the function value information from all subregions and, when it has all the information, it calculates the cut off level and sends it back to all the processors.

This organization makes it clear that our concurrent algorithm requires very little shared information, and therefore is well suited for implementation on a local memory multiprocessor. If a local memory multiprocessor is used, at each iteration the number of messages received and sent by each subregion process will be two plus the number of local searches conducted by that processor, plus one more if sample reduction is applied. At each iteration the master process will receive, and send, $2P$ messages ($3P$ with sample reduction) plus the total number of local searches for that iteration. The messages all are short, containing either one number, one n-vector, or a small number of n-vectors. Thus the total interprocess communication requirements are quite small.

Finally, we will examine how much overhead is introduced by the parallelization of the algorithm at Steps 1, 2 and 3 of Algorithm 3.1, and how fully we expect all processors to be utilized.

In Step 1 each processor samples N/P points and evaluates the function at each one of them. This step requires no interprocess communication or parallel overhead, and is expected to achieve equal utilization of all processors as long as the time required to evaluate $f(x)$ at different points x is (nearly) uniform.

Now consider the start point selection step without sample reduction. Since the selection of candidate start points requires each processer to consider the same number of points (kN/P), we expect equal utilization of all processors during this portion of Step 2. In our experience, the second part of the step, in which the border points are resolved, requires very little running time in comparison. So in Step 2 we introduce little parallel overhead or interprocess communication (each process has to send and read one message), and as in Step 1 we expect all processors to do the same amount of work. In fact, we will see in Section 5 that Step 2 has the interesting effect of applying a divide and conquer strategy that actually causes greater than linear speedup in comparison to some standard sequential implementations.

If Step 2 is run with a cut off level, however, the distribution of work among the processors may no longer be uniform. This is because the reduced sample points may not be equally distributed among the subregions and therefore the reduced sample size per processor may differ substantially. Processors handling a subregion with a relatively small reduced sample size will complete the selection of candidate start points faster than processors with a large reduced sample. These processors will then be idle until all processors have finished selecting their candidate start points and the border resolution phase can be started. The effects of this imbalance will be seen in some of the computational results in Section 5.

Recall, however, from the discussion at the beginning of this section, that the imbalance in utilization of processors in Step 2 caused by the sample reduction phase becomes unimportant as the cost of function evaluations rise. This is because the costs of Steps 1 and 3 then dominate the running time, and the cost of Step 2, which involves no function evaluations, becomes insignificant. This phenomenon is reflected in the simulated results for expensive functions given in Section 5.

The imbalance in Step 2 caused by sample reduction also would be less important if the processors early finishing could be employed in some other useful manner. To some extent this seems possible. If the subregion handled by the early finishing processor contains a candidate start point that is not within the critical distance of any subregion border, the processor could avoid being idle by starting a local search from this point immediately. This possibility is also examined in Section 5.

Finally, consider Step 3, the local minimization phase. Again, the cost of interprocessor communication in this phase is small (one message sent and received for each local search) and no other parallel overhead is introduced. As we have discussed previously, however, this step will probably not utilize all processors evenly due to the uneven lengths of local searches, and the fact that the number of searches may be less than the number of processors. This is one of the main effects that we will examine in Section 5. We will also present some results about improving the efficiency of the local minimization phase by introducing concurrency into the individual local minimizations.

In summary, the concurrent global optimization algorithm adds few new costs, either new operations or interprocess communication, to those present in the sequential algorithm. In the sampling phase, and the start point selection phase if run without sample reduction, it appears to readily allow full utilization of all processors. In the local minimization phase, and the start point selection phase if sample reduction is used, the synchronization requirements may cause some processors to be idle at some times. In problems where function evaluation is expensive, the expense of the algorithm is dominated by the sampling and local minimization steps, so that the efficiency of these steps is most important and the start point selection step becomes relatively unimportant anyhow.

4. Theoretical properties of concurrent multi-level single linkage

Although the concurrent and sequential multi-level single linkage algorithms are very similar in terms of the sampling and searching they perform, the assumptions under which the theoretical properties of the sequential multi-level single linkage algorithm are proven do not all hold for the concurrent method. In particular, the analysis of the sequential method assumes that the sample points are drawn from a uniform distribution over the feasible region S. In the concurrent algorithm this is no longer the case; instead, the sample points are generated from uniform distributions over the subregions.

The analysis of the sequential multi-level single linkage algorithm can be adapted, however, to show that the two properties mentioned in Section 2 also hold for the concurrent algorithm we discussed in Section 3. First, if the critical distance tends to 0 with increasing k, then with probability 1 all isolated local minima with values below the cut off level will be identified in a finite number of iterations. Second, if in (2.1) $\sigma > 4$, then if sampling continues forever, the number of local searches will be finite with probability 1. The proofs for these results involve appropriate modifications of the proofs for the sequential case; we refer to (Dert (1986)) for the details.

As in the sequential case, it is also possible to carry out a theoretical analysis of the expected running time of the algorithm. Through the use of appropriate data structures (cf. Section 5.2), it can be shown that the sequential algorithm requires a computational effort that in expectation increases as a *linear* function of the sample size. This result continues to hold for the concurrent version, even if a cut off level has to be computed centrally. A crucial role in the required computation of every gth value is then played by a dynamic selection method by Postmus, Rinnooy Kan and Timmer (1983), whose expected running time has very attractive properties. Again, we refer to (Dert (1986)) for full details.

5. Computational testing

We have implemented and tested the concurrent global optimization algorithm described in Section 3 on a network of computer workstations. This section reports the results of these tests. First, we briefly describe our parallel computing environment.

5.1. The testing environment

The University of Colorado is engaged in a large research project on the use of a network of computer workstations for concurrent computation. This project includes developing and implementing numerical algorithms for important practical problems that are well suited to this loosely coupled multiprocessing environment. It also includes the development of systems and software support that will make a network of computers easier to use for distributed concurrent computation. This project is supported by a Coordinated Experimental Research grant from the National Science Foundation as well as individual research grants. The concurrent global optimization algorithm described in Section 3 is an excellent candidate for solution in this environment.

Our current test environment consists of a network of Sun workstations, connected on an ethernet and sharing several file servers. The experiments reported in Section 5.2 were conducted on a dedicated subnet consisting of four or eight Sun-3 workstations. That is, when we conducted these experiments we were the only users of these workstations and the subnet was physically disconnected from the remainder of our computer network. Thus, the subnet functioned as a dedicated local memory multiprocessor.

Our ability to use the network of workstations for distributed concurrent processing is based upon the Sun version of the Berkeley Unix 4.2 operating system, which each workstation runs. The Berkeley Unix 4.2 operating system provides the basic interprocessor communication facility, the ability to send messages between processes on different machines, that is needed to use a network of computers as a multi-processor. In Berkeley Unix 4.2, this capability is provided by stream sockets, reliable point to point connections between two processors, as well as datagram sockets. When combined with the Unix *fork* and *exec* commands, these facilities allow a process on one computer to start a process on another computer and subsequently to communicate with it.

Researchers in the Computer Science Department at the University of Colorado have built a distributed processing utilities package, called DPUP, that makes a network of computers running the Berkeley Unix 4.2 operating system easier to use for distributed concurrent processing (Gardner et al. (1986)). DPUP builds upon the interprocessor communication facilities in Berkeley 4.2 to provide two models of concurrent computation. The first is a master-slave model where all processes are linked to one master in a "spokes of a wheel" arrangement and all communication is through the master. The second is a broadcast model where each process is an equal member of a ring of processes and can send messages to all of the processes at once. For both models, DPUP provides several basic concurrency capabilities including the creation and termination of remote processes (with required communication connection automatically established) and various means to send and receive messages. Our concurrent global optimization software uses the master-slave model of DPUP.

Our parallel algorithm has recently been ported to the Intel hypercube and relatively little difficulty (see Eskow and Schnabel (1987)). As expected, due to the small amount of synchronization and communication required, the performance is quite similar to that on the network of workstations. We would expect similar behavior on almost any MIMD computer. As noted in Eskow and Schnabel (1987), once each function evaluation requires about 10 000 floating point operations, the cost of function evaluations will swamp all other costs of the parallel algorithm, including communication, and so our expensive function results (Tables 5.6, 5.12) will be indicative of the performance on any MIMD computer.

5.2. Computational results

Both the sequential and the concurrent global optimization algorithms have been run on the test problems given in Dixon and Szego (1978), and on some problems from Levy and Gomez (1985). At present these seem to be among the few widely accepted global optimization test problems. We comment first on our results on the Dixon–Szego test set, and then more briefly on our results on the Levy–Gomez problems.

The characteristics of the problems from Dixon and Szego (1978) are summarized in Table 5.1. The problems are low dimensional (up to 6 variables) with only few local minimizers (up to 10). In addition, evaluation of the test functions is very cheap. These characteristics limit what one can determine from the test set, and how one should interpret the computational results, in several ways.

Table 5.1
Test problem data, Dixon–Szego problems

Problem name	Abbreviation	Number of variables	Number of local minimizers
Goldstein–Price	GP	2	4
Branin	BR	2	3
Hartman 3	H3	3	4
Hartman 6	H6	6	4
Shekel 5	S5	4	5
Shekel 7	S7	4	7
Shekel 10	S10	4	10

The small number of variables and local minimizers limits the amount of parallelism that can be obtained in solving the global optimization problem. The number of sample points required, the number of local searches required, and often the number of iterations required would all be significantly higher for more difficult problems, which in turn would enable the use of more concurrency. The Levy–Gomez problems have considerably more local minimizers, but the number of variables still is small.

The fact that the evaluation of the test functions themselves is very cheap (sometimes requiring only a few floating point operations) means that our timing results are not indicative of performance on many real world problems where function evaluation is the dominant cost. Therefore we will report two speedup measures. The first measure is the actual timed speedup, the time required by the sequential algorithm to solve a problem divided by the time required by the concurrent algorithm to solve the same problem, i.e.

$$\text{speedup}_{\text{timed}} = \frac{\text{elapsed time sequential algorithm}}{\text{elapsed time concurrent algorithm}}. \tag{5.1}$$

In many of our experiments this measure is dominated by the start point selection phase, which in fact requires no function evaluations. Thus this measure is an interesting indication of how well we have sped up the overhead calculations of the algorithm, and also gives some indication of how practical our approach would be on small problems with very inexpensive function evaluations.

In many practical optimization problems, however, the evaluation of the objective function $f(x)$ is very expensive. The computational effort then will consist mainly of computing function values. So in this case one is primarily interested in the distribution of function evaluations among the processors. To use our test results to indicate the speedup our concurrent algorithm would achieve on this type of problem, we introduce a second speedup measure. Let $\Psi_{i,j}$ denote the number of function evaluations done by the ith processor at the jth iteration, and let the total number of processors and iterations be P and I, respectively. Then the speedup for expensive function evaluations may be approximated by

$$\text{speedup}_{\text{expensive func}} = \frac{\sum_{j=1}^{I} \sum_{i=1}^{P} \Psi_{i,j}}{\sum_{j=1}^{I} \max_{(k=1,\dots,P)} \Psi_{k,j}}. \tag{5.2}$$

This measure is the limit of the timed speedup ratio we would obtain on our test problems if the function values were unchanged but the cost of each function evaluation was increased without bound. Contrary to the first measure, it is independent of the speedup achieved during the start point selection phase.

Our test results are presented for four different modes of operation of the sequential and concurrent algorithms: using 200 or 1000 sample points per iteration, and with and without sample reduction. When sample reduction was used, the sample size was reduced to $100k$ points, where k is the iteration number. The parameter σ was set to 4 in all cases. Since the algorithm is stochastic, the results are influenced to some extent by the random sample that is generated. To dampen the effect of the variation in random samples, 10 independent runs were performed for each problem/algorithm combination.

The reliability of each algorithm on each problem is summarized in Table 5.2, and the average costs in function evaluations are given in Table 5.3. (These data are given for the 8 processor concurrent algorithm but are very similar when using different numbers of processors.) On these simple test problems, using 200 sample points per iteration usually led the algorithm to require fewer total function evaluations, although the reliability of the algorithm was somewhat better with 1000 points per iteration. (The reliability results are similar to those reported in Rinnooy Kan and Timmer (1985a) and no attempt was made to change the algorithm to improve upon them.) We consider the 1000 point per iteration size, however, to be far more indicative of what would be required on problems with more variables or

Table 5.2

Number times global minimizer found in 10 runs, Dixon–Szego problems (8 processors)

Problem:	GP	BR	H3	H6	S5	S7	S10
200 points per iteration, no sample reduction	10	10	10	10	8	6	6
1000 points per iteration, no sample reduction	10	10	10	10	10	7	7
200 points per iteration, sample reduction to 100	10	10	10	10	8	6	6
1000 points per iteration, sample reduction to 100	10	10	10	10	10	7	7

Table 5.3

Number of function evaluations, averaged over 10 runs, Dixon–Szego problems (8 processors)

Problem:	GP	BR	H3	H6	S5	S7	S10
200 points per iteration, no sample reduction	412	306	380	1522	487	469	447
1000 points per iteration, no sample reduction	1420	1150	1213	3104	1395	1346	1375
200 points per iteration, sample reduction to 100	376	261	307	658	327	330	327
1000 points per iteration, sample reduction to 100	1112	1064	1161	1972	1177	1203	1884

more local minimizers. Therefore we consider the test results with 1000 sample points per iteration to be the more important ones. In general, using sample reduction appears to lead to a more efficient algorithm. As discussed in Section 3.4, our concurrent start point selection algorithm may incur significant idle time when used with sample reduction. This affects our timing results where the start point selection has a significant impact, but not the expensive function evaluation results where start point selection is irrelevant.

When we first timed our concurrent global optimization algorithm (without sample reduction) on a network of 3 Sun-2 workstations, the speedups in comparison to the sequential algorithm on a Sun-2 were consistently *greater than* 3. In fact, they generally ranged from about 3 to about 8. The reason for this was fairly easy to see. A large portion of the time was being spent in the start point selection phase, which in the sequential case used an $O(N^2)$ algorithm, where N is the number of sample points. The concurrent algorithm was essentially applying one stage of divide and conquer to this algorithm, first dividing the process into 3 equal parts (which reduces the *total* work of an $O(N^2)$ algorithm by a factor of 3 and thus would induce a speedup of 9 in our situation), and then applying the border resolution strategy to patch together the 3 regions. But since the border resolution strategy is only applied to a small portion of the original sample, its cost is small and the total speedup for the start point phase was still close to 9. Indeed, when we modified the sequential algorithm to use the identical strategy, that is divide the feasible region into 3 equal parts, do the start point selection in each separately, and then do border resolution, the sequential algorithm times dropped significantly and the speedups by the concurrent algorithm no longer were greater than 3. Instead, they ranged from about 2 to 3.

From a theoretical point of view, it is known that one can do better than the straightforward $O(N^2)$ algorithm for start point selection. Timmer (1984) shows that the *spiral search* technique of Bentley, Weide and Yao (1980) can be applied so that the expected running time of the start point selection phase, when totaled over all the iterations of the algorithm, is linear in the total number of sample points used. We subsequently implemented this technique in the manner suggested by Timmer. We found that it is more efficient than the one stage divide and conquer strategy described above only for problems of very small dimension. For example, with sample size 1000 and $n = 4$, the time required by a 16 subdivision divide and conquer algorithm is roughly equivalent to that required by spiral search, while when $n = 6$ a 4 subdivision divide and conquer algorithm already is about as efficient as spiral search and a 16 subdivision divide and conquer is about 4 times more efficient. Thus for our computational results on sequential machines, we have chosen to use the one stage divide and conquer approach: when comparing to a P processor concurrent algorithm, we use a sequential algorithm that also subdivides into P subregions in the start point selection phase. (This accounts for the different times for the same sequential algorithms in Tables 5.4 and 5.5.) Note that from a computational point of view, our research into concurrent global optimization algorithm seems to have led to an improved sequential algorithm as well.

Tables 5.4 and 5.5 give the timed speedups of our concurrent global optimization algorithm using 4 and 8 processors, respectively. The differences between the times for the same sequential algorithms in the two tables shows that they are dominated by the start point selection phase. In the two cases without sample reduction, there often is almost a factor of two difference in the sequential times between Table 5.4 and 5.5. This is accounted for by the factor of nearly 2 reduction in the start point selection phase when switching from the 4 subdivision to 8 subdivision sequential algorithm, as discussed above; the times required by all other phases of the sequential algorithm are identical in the two cases but take a small portion of the total.

The times for 4 processors without sample reduction show good speedup. This is especially true with sample size 1000 where the speedups average about 3.6. In this case the start point selection phase is dominant and is parallelized almost fully. Recall that this sample size is more indicative of the sample size that would be used on most real-world problems. For sample size 200 the speedups are somewhat less good, averaging about 2.8. Here the total running time has become small enough

Table 5.4

Times and speedups on 4 processors, averaged over 10 runs, Dixon–Szego problems (times in seconds)

Problem:	GP	BR	H3	H6	S5	S7	S10
200 points per iteration, no sample reduction							
Sequential Time	4.7	5.0	9.0	31.2	9.3	10.1	10.9
Concurrent Time	1.6	1.7	3.2	10.0	3.7	3.8	3.6
Speedup	2.9	3.0	2.8	3.1	2.5	2.7	3.0
1000 points per iteration, no sample reduction							
Sequential Time	142.2	152.1	184.0	339.0	186.1	189.0	188.0
Concurrent Time	38.0	40.9	55.8	98.8	51.3	51.2	51.1
Speedup	3.7	3.7	3.3	3.4	3.6	3.7	3.7
200 points per iteration, sample reduction to 100							
Sequential Time	1.6	1.8	4.2	16.9	3.2	3.7	4.3
Concurrent Time	1.0	1.1	1.7	5.3	1.5	1.6	1.7
Speedup	1.6	1.6	2.5	3.2	2.1	2.3	2.5
1000 points per iteration, sample reduction to 100							
Sequential Time	4.0	4.0	9.1	24.3	6.3	7.5	17.0
Concurrent Time	1.8	2.0	3.7	7.9	2.5	2.8	5.2
Speedup	2.2	2.0	2.5	3.1	2.5	2.7	3.3

that the idle time in the search phase and the small communications overhead begins to have an effect. When sample reduction is used, the speedups are a little lower still, averaging about 2.3 and 2.5 for the two sample sizes. This reduction in speedup is caused mainly by the inefficiency in start point selection phase when sample reduction is used; the various subregions often turn out to have quite different reduced sample sizes and thus all but one processor must wait until the processor with the most reduced sample points selects its candidate start points. Recall that since the start point selection phase requires no function evaluations, the algorithmic aspects that mainly determine these timing results will be irrelevant in the expensive function results.

The speedups for 8 processors are still quite good with 1000 sample points and no sample reduction, averaging about 6.2. By comparison to the 8 processor line for this algorithm in Table 5.6, it is seen that these speedups are fairly close to the expensive function limits. The reason is that with 8 processors, the start point selection time per processor has become relatively small and the sampling and search phases are beginning to dominate. The limit in the parallelism is caused by the small total number of local searches (usually there are fewer than 8) and the unequal lengths of the searches.

The speedups for the other algorithms with 8 processors are not very good, averaging 3.0 for 200 points without sample reduction and 2.3 and 3.1 for the two

Table 5.5
Times and speedups on 8 processors, averaged over 10 runs, Dixon–Szego problems (times in seconds)

Problem:	GP	BR	H3	H6	S5	S7	S10
200 points per iteration, no sample reduction							
Sequential Time	2.7	3.1	5.1	27.7	6.0	6.5	6.9
Concurrent Time	1.2	1.2	1.7	5.8	2.2	2.3	2.2
Speedup	2.3	2.6	3.0	4.8	2.7	2.8	3.1
1000 points per iteration, no sample reduction							
Sequential Time	68.3	74.0	76.7	223.0	94.7	94.8	97.6
Concurrent Time	11.7	10.9	12.3	40.7	51.1	15.2	14.5
Speedup	5.8	6.8	6.2	5.5	6.3	6.2	6.7
200 points per iteration, sample reduction to 100							
Sequential Time	1.5	1.5	3.7	15.7	3.0	3.6	3.8
Concurrent Time	1.1	1.0	1.8	3.5	1.5	1.6	1.7
Speedup	1.4	1.5	2.1	4.5	2.0	2.3	2.2
1000 points per iteration, sample reduction to 100							
Sequential Time	4.0	4.0	9.1	23.5	6.3	7.6	17.4
Concurrent Time	1.9	2.0	2.9	5.5	2.0	2.4	4.5
Speedup	2.1	2.0	3.1	4.3	3.2	3.2	3.9

algorithms with sample reduction. The main reason for these results is that the run times are so small that the interprocessor communication overhead and the idle times in the search phase have a large effect. Indeed, in the case of 200 points with sample reduction, the run times with 8 processors are essentially the same as with 4 processors. A careful breakdown of these times showed that the increase in interprocess communication times when going from 4 to 8 processors was a few tenths of a second, and offset the small decreases that were possible in the sampling, start point selection, and search phases of the algorithm. This demonstrates the limit to the grain of parallelism that is effective in our multicomputer multiprocessing environment, and is simply a consequence of the very inexpensive function evaluations.

The expensive function evaluation speedups for each algorithm are given in Table 5.6. Note that once we have run our global optimization algorithm on a particular problem with any particular number of processors, we know the total number of iterations it will use, the total number of sample points it will use, and the number and length of local searches it will perform at each iteration, regardless of the number of processors. (There can be slight variations due to the stochastic effects and the effect of requiring an equal number of sample points per subregion.) Thus, given the rule for ordering and distributing local searches in the concurrent algorithm, we can calculate the expensive function speedup on this problem for any other

Table 5.6

Simulated speedups for expensive function evaluations, averaged over 10 runs, Dixon–Szego problems

	Number of processors	Problem: GP	BR	H3	H6	S5	S7	S10
200 points per iteration, no sample reduction	2	1.9	1.9	1.9	2.0	1.8	1.8	1.9
	4	3.4	3.5	3.3	3.6	2.7	3.0	3.1
	8	4.6	5.0	4.8	6.1	3.5	3.9	4.1
	16	5.4	6.4	5.7	8.6	4.1	4.6	4.8
	32	5.9	7.4	6.3	9.6	4.4	5.1	5.3
	64	6.1	8.0	6.6	9.8	4.6	5.3	5.5
	200	6.3	8.4	6.9	10.0	4.8	5.5	5.7
1000 points per iteration, no sample reduction	2	2.0	2.0	2.0	2.0	2.0	2.0	2.0
	4	3.9	3.9	3.8	3.8	3.6	3.6	3.7
	8	7.1	7.1	6.7	6.8	6.0	6.3	6.5
	16	11.0	11.2	10.3	11.7	8.9	9.4	9.5
	32	14.6	15.9	14.0	15.4	12.0	12.5	12.4
	64	17.5	20.2	17.3	16.7	14.6	15.2	14.9
	1000	21.4	27.0	21.6	18.1	18.3	18.7	17.9
200 points per iteration, sample reduction to 100	2	1.9	1.9	1.9	1.9	1.9	1.9	1.9
	4	3.3	3.4	3.0	3.2	3.0	3.3	3.1
	8	4.4	5.0	4.0	4.3	4.1	5.1	4.1
	16	5.1	6.7	4.8	4.9	4.9	6.6	5.0
	32	5.6	7.9	5.3	5.1	5.4	7.1	5.6
	64	5.9	8.8	5.6	5.3	5.7	7.4	5.9
	200	6.1	9.4	5.9	5.3	6.0	7.7	6.2
1000 points per iteration, sample reduction to 100	2	2.0	2.0	2.0	2.0	2.0	2.0	2.0
	4	3.8	3.8	3.8	3.7	3.8	3.8	3.8
	8	6.8	6.9	6.5	6.5	6.8	6.7	6.8
	16	11.1	11.8	10.1	9.7	10.6	10.4	11.0
	32	16.2	18.0	13.9	11.4	14.7	14.2	15.7
	64	21.4	24.8	17.2	12.6	18.4	17.5	20.3
	1000	29.9	37.0	21.7	13.8	23.7	22.0	27.7

number of processors from eq. (5.2). We use this flexibility to calculate expensive function speedups, for each problem and algorithm, for 2, 4, 8, 16, 32, and 64 processors. (The data from the 8 processor concurrent algorithm is used.) In addition, the speedup with the number of processors equal to the number of sample points per iteration, 200 or 1000, is given. Since the number of local searches per iteration is always far less than 200, the cost of an iteration of the concurrent algorithm in this case is 1 (for the sampling phase) plus the number of function evaluations in the longest local search. This is the limiting speedup for this algorithm; i.e. if more processors were available the expensive function speedup would not change.

For the algorithms with 1000 sample points per iteration, the expensive function speedups are reasonably high, whether or not sample reduction is used. With 8 processors most problems make at least 80% utilization of the processors, with 16 processors the utilization is generally at least 60%, and even with 32 processors the utilization is often around 50%. The limiting speedups (for 1000 processors) are over 20 in most cases. Since the maximum parallelism in the local search phase is generally between 5 and 10, the sampling phase is having a large effect; in the problem with the longest searches, Hartman 6, the speedups are lower. The effect of sample reduction on the speedups is uneven; reducing the sample tends to reduce the number of local searches which reduces parallelism, but the length of the longest search sometimes decreases which increases the speedup.

When 200 sample points per iteration are used, the expensive function speedups are much lower, with limiting speedups between 5 and 10. In this case the sampling phase is a much smaller portion of the algorithm and the local search phase dominates. The speedups are simply a reflection of the small number of local searches. The concurrency in this case could be improved by parallelizing the individual local searches; this is discussed briefly in Section 6.

We have also tested our parallel global optimization algorithm on four of the test problems from Levy and Gomez (1985). These problems differ from those in Dixon and Szego (1978) in that the number of local minimizers is much larger. The number of variables still is small, and the cost of function evaluation still is very low. The characteristics of the problems we have tested are summarized in Table 5.7.

Table 5.7
Test problem data, Levy–Gomez problems

Problem number (Levy–Gomez (1985))	Number of variables	Number of local minimizers
3	2	760
7	2	25
8	3	125
9	4	625

It is likely that the stochastic methods described in this paper are not the best methods for problems with hundreds or thousands of minimizers, because they are oriented towards finding all the local minimizers, or at least all below a certain cutoff level. In addition, the stopping rule that is used in the stochastic methods, derived from (2.2), requires that the sample size be greater than $2w^2$, where w is the number of local minimizers found, and this may be too large when there are very many local minimizers. Research still is necessary to construct a global optimization method that is efficient on such problems and also has strong theoretical properties. Our primary interest in this paper, however, is not in constructing such a method, but rather in studying how effectively the sequential stochastic approach can be adapted to parallel computers, and how this depends on the problem characteristics. Therefore we have applied the algorithms described in Sections 2 and 3 to the Levy–Gomez problems, with one modification. Rather than satisfy the standard stopping rule, we have simply asked our algorithm to do the same amount of work as for almost all the problems in the first test set, that is one iteration with either 200 or 1000 sample points, either with or without sample reduction. Table 5.8 shows that the algorithm still found the global minimizer in 79 of our 80 test runs, so that additional iterations would only have served to confirm the global minimizer. Table 5.9 shows that the average number of function evaluations is higher than for the Dixon–Szego test problems; this is due to the large number of local searches conducted, a consequence of the larger number of local minimizers. Due to the greater expense of these test problems, we made 5 rather than 10 runs for each case.

Our test results on the Levy–Gomez problems are summarized in Tables 5.10–5.12. In general they are very similar to the test results for the Dixon–Szego test set. The timing results for 4 and 8 processors are given in Tables 5.10 and 5.11, respectively. On the average, the speedups without sample reduction are 10–20% lower than for the Levy–Gomez test set; this is due to the algorithm spending a higher proportion of its time in the local search phase, which does not parallelize perfectly, and a

Table 5.8
Number times global minimizer found in 5 runs, Levy-Gomez problems

Problem No.:	3	7	8	9
200 points per iteration, no sample reduction	5	5	5	5
1000 points per iteration, no sample reduction	5	5	5	5
200 points per iteration, sample reduction to 100	5	5	5	4
1000 points per iteration, sample reduction to 100	5	5	5	5

Table 5.9
Number of function evaluations, averaged over 5 runs, Levy-Gomez problems (8 processors)

Problem No.:	3	7	8	9
200 points per iteration, no sample reduction	519	422	465	1116
1000 points per iteration, no sample reduction	1804	1702	2756	3837
200 points per iteration, sample reduction to 100	470	314	274	831
1000 points per iteration, sample reduction to 100	1478	1194	1769	1988

Table 5.10

Times and speedups on 4 processors, averaged over 5 runs, Levy-Gomez problems (times in seconds)

Problem No.:	3	7	8	9
200 points per iteration, no sample reduction				
Sequential Time	9.2	5.2	8.9	14.7
Concurrent Time	3.7	2.1	3.7	6.1
Speedup	2.5	2.5	2.4	2.4
1000 points per iteration, no sample reduction				
Sequential Time	69.6	105.2	135.4	190.8
Concurrent Time	21.5	30.4	45.9	62.0
Speedup	3.2	3.5	3.0	3.1
200 points per iteration, sample reduction to 100				
Sequential Time	9.0	3.5	5.6	9.7
Concurrent Time	3.5	1.5	2.4	3.9
Speedup	2.6	2.3	2.4	2.5
1000 points per iteration, sample reduction to 100				
Sequential Time	28.2	7.2	17.8	28.1
Concurrent Time	9.3	2.9	8.7	11.2
Speedup	3.0	2.5	2.0	2.5

correspondingly lower portion of its time in the sampling and start point phases, which parallelize better. When sample reduction is used, this effect is decreased, and in fact the average speedups for the Levy-Gomez problems are a bit higher than for the Dixon-Szego problems.

The timing results of Tables 5.10-5.11 are indicative of the performance of our algorithm when function evaluation is very inexpensive. Table 5.12 indicates what the performance of our algorithm would be on the Levy-Gomez problems if the function evaluations were expensive (say at least a few thousand floating point operations per function evaluation) but the problems were otherwise unchanged. Again, speedups for from 2 to 64 processors, as well as the limiting case of 200 or 1000 processors, are calculated. In general, the speedups are somewhat higher than for the Dixon-Szego problems in Table 5.6. This is primarily due to the larger number of local searches, which permit a more effective utilization of the processors during the local search phase.

Taken together, the results of this section confirm that our concurrent global optimization algorithm is able to exploit parallelism reasonably well, to the extent that it is available in the problem. The parallelism that is possible is limited by the number of variables, the number of local minimizers, and the expense of the function evaluations. It appears that it is possible to obtain good utilization of a moderate

Table 5.11

Times and speedups on 8 processors, averaged over 5 runs, Levy-Gomez problems (times in seconds)

Problem No.:	3	7	8	9
200 points per iteration, no sample reduction				
Sequential Time	8.7	3.8	6.2	14.1
Concurrent Time	3.8	1.4	2.4	4.8
Speedup	2.3	2.7	2.6	2.9
1000 points per iteration, no sample reduction				
Sequential Time	52.5	49.6	86.0	121.0
Concurrent Time	14.1	7.2	17.0	24.8
Speedup	3.7	6.9	5.1	4.9
200 points per iteration, sample reduction to 100				
Sequential Time	8.9	3.1	4.5	11.8
Concurrent Time	3.5	1.3	1.9	5.6
Speedup	2.5	2.4	2.3	2.1
1000 points per iteration, sample reduction to 100				
Sequential Time	24.6	7.3	16.7	30.7
Concurrent Time	6.2	2.3	4.7	6.8
Speedup	4.0	3.1	3.6	4.5

number of processors even for problems with inexpensive function evaluations, and that quite effective use of multiple processors can be made for problems with expensive function evaluations. As expected, the speedups for problems with expensive function evaluations are higher when the number of local minimizers and searches is higher. Besides the inherent limitations of the problem sizes, the main deterents from achieving even higher speedups with the current algorithm are the sequential nature of the individual local searches, and the imbalance in the start point selection phase when sample reduction is used. In the next section we briefly discuss several techniques for introducing additional concurrency into the global optimization algorithm that deal with these problems.

6. Future research directions

The concurrent global optimization algorithm discussed in this paper appears to have two important limitations. First, the amount of parallelism is limited by the number of local minimizations at each iteration and possibly by the distribution of their lengths. Secondly, the sampling effort is distributed uniformly over the entire feasible region regardless of the characteristics of the objective function. (This

Table 5.12

Simulated speedups for expensive function evaluations averaged over 5 runs, Levy–Gomez problems

	Number of	Problem No.:			
	processors	3	7	8	9
200 points per iteration, no sample reduction	2	2.0	1.9	1.8	1.9
	4	3.7	3.6	3.3	3.6
	8	6.7	6.2	4.2	5.4
	16	9.8	9.5	4.8	5.9
	32	11.1	11.2	5.1	6.1
	64	11.8	12.1	5.3	6.2
	200	12.4	12.9	5.4	6.3
1000 points per iteration, no sample reduction	2	2.0	2.0	2.0	2.0
	4	3.9	3.9	3.8	3.8
	8	7.7	7.4	7.4	7.2
	16	14.2	13.7	12.7	12.0
	32	22.8	23.6	18.4	15.2
	64	28.6	31.2	20.7	16.3
	1000	36.8	42.1	23.1	17.4
200 points per iteration, sample reduction to 100	2	1.9	1.9	1.6	1.9
	4	3.7	3.6	2.5	3.4
	8	6.7	5.6	3.2	4.9
	16	9.1	7.3	3.8	5.3
	32	10.4	8.5	4.2	5.5
	64	11.1	9.3	4.4	5.6
	200	11.7	9.9	4.5	5.6
1000 points per iteration, sample reduction to 100	2	2.0	2.0	2.0	2.0
	4	3.9	3.8	3.8	3.8
	8	7.5	7.3	7.1	6.4
	16	13.9	12.8	11.4	8.5
	32	20.3	19.3	14.4	9.8
	64	26.0	26.0	16.5	10.7
	1000	34.6	37.6	19.1	11.6

limitation is shared by the sequential algorithm.) We are working on ways to overcome both limitations.

When function evaluation is expensive, one way to introduce more parallelism into the local minimization phase of our algorithm is to distribute the gradient calculations (assuming they are done by finite differences as is often the case) among various processors. This has the potential to increase the parallelism obtained by up to a factor of n, although a speedup of about $n/2$ is the most one can expect if each gradient evaluation is preceded by a function evaluation that is performed sequentially. If there are not enough processors to evaluate n function values for each local search simultaneously, then a strategy for ordering function and gradient evaluations is required, and this strategy may also be used to attempt to give the longer local searches higher priority. We have conducted some preliminary tests of such a strategy on the test problems of Section 5. Our indications are that it can significantly increase the speedup of the local minimization phase of each iteration, if function evaluation is expensive and the number of processors significantly exceeds the number of local searches for that iteration. Further modifications, including conducting function and gradient evaluations simultaneously before it is known whether the gradient actually is needed and using multiple gradient values at an iteration, are currently being investigated in the contexts of local and global minimization.

A second approach for obtaining more parallelism in global optimization is to move towards a more asynchronous algorithm. By this we mean that there are fewer synchronization points where a task on one processor cannot start until a task on another processor has completed. In our concurrent global optimization algorithm, these synchronization points occur at the end of the start point selection phase before the local minimization phase can begin, and at the end of the local minimization phase before the next iteration can begin. One way to reduce synchronization within the framework of a concurrent multi-level single-linkage algorithm is to make each subregion autonomous, meaning that it decides on its own to conduct searches or whether to go on to the next iteration. Instead of the synchronization in our present algorithm, in this approach a scheduler process distributes the local minimization tasks and sampling/start point selection tasks that are generated. This approach also naturally permits work to be concentrated in subregions of greatest interest. We currently are developing such an algorithm.

Finally, there is an important need for more difficult global optimization test problems, especially test problems drawn from actual applications.

Acknowledgement

We thank Betty Eskow for her extensive help in continuing computer testing of our parallel algorithms after Cees Dert returned to the Netherlands. This included all of the testing of the Levy-Gomez problems.

References

R.S. Anderssen, "Global optimization," in: R.S. Anderssen, L.S. Jennings and D.M. Ryan, eds. *Optimization* (University of Queensland Press, 1972).

C.G.E. Boender, "The generalized multinormal distribution: a Bayesian analysis and applications," Ph.D. thesis, Econometric Institute, Erasmus University (Rotterdam, The Netherlands, 1984).

C.G.E. Boender and A.H.G. Rinnooy Kan, "Bayesian stopping rules for a class of stochastic global optimization methods," Technical Report, Erasmus University (Rotterdam, The Netherlands, 1983).

C.G.E. Boender, A.H.G. Rinnooy Kan, L. Stougie and G.T. Timmer, "A stochastic method for global optimization," *Mathematical Programming* 22 (1982) 125-140.

F.H. Branin, "Widely convergent methods for finding multiple solutions of simultaneous nonlinear equations," *IBM Journal of Research Developments* (1972) 504-522.

F.H. Branin and S.K. Hoo, "A method for finding multiple extreme of a function of n variables," in: F.A. Lootsma, ed., *Numerical Methods of Nonlinear Optimization* (Academic Press, London, 1972).

S.H. Brooks, "A discussion of random methods for seeking maxima," *Operations Research* 6 (1958) 244-251.

J.E. Dennis Jr. and R.B. Schnabel, *Numerical Methods for Nonlinear Equations and Unconstrained Optimization* (Prentice-Hall, Englewood Cliffs, New Jersey, 1983).

C.L. Dert, "A parallel algorithm for global optimization," Masters thesis, Econometric Institute, Erasmus University (Rotterdam, The Netherlands, 1986).

L. Devroye, "A bibliography on random search," Technical Report, McGill University (Montreal, 1979).

L.C.W. Dixon and G.P. Szego, eds., *Towards Global Optimization* 2 (North-Holland, Amsterdam, 1978).

E. Eskow and R.B. Schnabel, "Using mathematical modeling to aid in parallel algorithm development," *Proceedings of Third SIAM Conference on Parallel Processing for Scientific Computation* (Los Angeles, 1987).

J.B.E. Frenk and A.H.G. Rinnooy Kan, "The asymptotic optimality of the LPT rule," *Mathematics of Operations Research* (to appear).

T.J. Gardner, I.M. Gerard, C.R. Mowers, E. Nemeth and R.B. Schnabel, "DPUP: A distributed processing utilities package," Technical Report CU-CS-337-86, University of Colorado, Department of Computer Science (Boulder, Colorado, 1986).

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

A.A. Goldstein and J.F. Price, "On descent from local minima," *Mathematics of Computation* 25 (1971) 569-574.

E.R. Hansen, "Global optimization using interval analysis—The multidimensional case," *Numerical Math* 34 (1980) 247-270.

E.R. Hansen and S. Sengupta, "Global constrained optimization using interval analysis," in: K. Nickel, ed., *Interval Mathematics* (Academic Press, London, 1980).

A.V. Levy and S. Gomez, "The tunneling method applied to global optimization," in: P.T. Boggs, R.H. Byrd and R.B. Schnabel, eds., *Numerical Optimization* 1984 (SIAM, Philadelphia, 1985) 213-244.

A.V. Levy and A. Montalvo, "The tunneling algorithm for the global minimization of functions," *SIAM Journal on Scientific and Statistical Computing* 6 (1985) 15-29.

J.T. Postmus, A.H.G. Rinnooy Kan and G.T. Timmer, "An efficient dynamic selection method," *Communications of the ACM* 26 (1983) 878-881.

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

A.H.G. Rinnooy Kan and G.T. Timmer, "Stochastic methods for global optimization," *American Journal of Mathematical and Management Sciences* 4 (1984) 7-40.

A.H.G. Rinnooy Kan and G.T. Timmer, "A stochastic approach to global optimization," in: P. Boggs, R. Byrd and R.B. Schnabel, eds., *Numerical Optimization* 1984 (SIAM, Philadelphia, 1985a) 245-262.

A.H.G. Rinnooy Kan and G.T. Timmer, "Stochastic global optimization methods—Part I: Clustering methods," Report 85391A, Econometric Institute, Erasmus University (Rotterdam, The Netherlands, 1985b).

A.H.G. Rinnooy Kan and G.T. Timmer, "Stochastic global optimization methods—Part II: Multi-level methods," Report 85401A, Econometric Institute, Erasmus University (Rotterdam, The Netherlands, 1985c).

C.L. Seitz, "The cosmic cube," *Communications of the ACM* 28 (1985) 22-33.

B.O. Shubert, "A sequential method seeking the global maximum of function," *SIAM Journal on Numerical Analysis* 9 (1972) 379-388.

F.J. Solis and R.J.E. Wets, "Minimization by random search techniques," *Mathematics of Operations Research* 6 (1981) 19-30.

G.T. Timmer, "Global optimization: A Bayesian approach," Ph.D. thesis, Econometric Institute, Erasmus University (Rotterdam, The Netherlands, 1984).

G.W. Walster, E.R. Hansen and S. Sengupta, "Test results for a global optimization algorithm," in: P. Boggs, R.H. Byrd and R.B. Schnabel, eds., *Numerical Optimization* 1984 (SIAM, Philadelphia, 1985) 272-287.

R. Zielinski, "A stochastic estimate of the structure of multi-external problems," *Mathematical Programming* 22 (1981) 104-116.