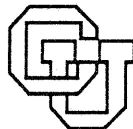
Adaptive, Asynchronous Stochastic Global Optimization Algorithms for Sequential and Parallel Computation

Sharon Smith,
Elizabeth Eskow,
and
Robert B. Schnabel

CU-CS-449-89 October 1989



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation.

The findings in this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

Abstract

We discuss new global optimization algorithms that are related to the stochastic methods of Rinnooy Kan and Timmer, and to our previous static, synchronous parallel version of this method. The new algorithms have two main new features. First, they adaptively concentrate the computation in the areas of the domain space that appear most likely to produce the global minimum. Secondly, on parallel computers, they use an asynchronous approach, combined with a central work scheduler, to avoid load balancing problems. We investigate several mechanisms for deciding when and how to make the adaptive adjustments. We also describe both algorithmic and implementation considerations involved in constructing the parallel asynchronous algorithm. Computational tests on sequential and parallel computers show that the adaptive and asynchronous features of our new method can substantially reduce the number of function evaluations, and the execution time, required by previous stochastic methods to solve global optimization problems.

1 Introduction

This paper presents new global optimization methods that were motivated by our consideration of parallel methods, but which are of interest for both sequential and parallel computation. The problem we consider is to find the lowest value of a nonlinear function, f, of n real variables, in a domain S of $\Re^n$ defined by upper and lower bounds on each variable. It is assumed that f may have multiple local minimizers, i.e. points z for which $f(x) \geq f(z)$ for all x in an open neighborhood containing z, and that the global minimizer is in the interior of S.

The global optimization problem is important and difficult, and effective methods for its solution have just begun to appear in recent years. These include deterministic methods such as the tunneling method [5], methods based on followed trajectories [1], and stochastic methods [6]. Since the problem is so computationally intensive, the advent of parallel computers has naturally led to interest in creating parallel methods for this problem as well.

In the past few years, we have been investigating *static, synchronous, stochastic parallel* methods for global optimization [2]. By a *static* algorithm, be it sequential or parallel, we mean one where the number of tasks, and usually their sizes, are known at the start of the computation. In particular, there is no attempt during the computation to adjust these parameters in order to achieve a more efficient solution. By a *synchronous* algorithm we mean one that periodically requires synchronization of the processors to satisfy precedence constraints among tasks or to update some global information. By *stochastic* we mean that the algorithm includes some random feature; the approach we take is based upon [6] and includes random sampling of the function in the domain space. Finally, our orientation towards *parallel* here is computation suited to any MIMD computer, i.e. a computer where multiple processors can perform independent computations concurrently. This includes shared memory multiprocessors, distributed memory multiprocessors such as hypercubes, and networks of computers used for concurrent computation.

In the course of implementing and testing the static synchronous parallel algorithm, we discovered several possibilities for its improvement. First, the synchronous approach sometimes led to load imbalance and poor processor utilization. This was because similar tasks that were performed in parallel, and were followed by a synchronization point, had widely varying execution times. Secondly, due to the static nature of the algorithm, insufficient effort was made upon focusing attention on the most fruitful areas of the domain space. This shortcoming was independent of whether the algorithm was sequential or parallel, although it was the partitioning of the domain space that was used in the parallel algorithm that made it natural to consider alternatives.

These problems have led us to consider alternative approaches to stochastic global optimization that have two new main features. First, they adaptively concentrate the computation in areas of the domain space that are most likely to produce the global minimum. Secondly, on parallel computers, they use an asynchronous approach to avoid load balancing problems. In the remainder of this section we motivate this approach by reviewing the stochastic approach of [6], our static, synchronous parallel version of it, and its performance and deficiencies. We also introduce the remedies to these deficiencies that are investigated in this paper.

1.1 Stochastic, static and synchronous global optimization algorithms

The global optimization algorithms described in this paper are all based upon the stochastic approach of [6]. These methods take an iterative approach. At each iteration, the function is evaluated at a number of randomly generated sample points in the domain space (typically 100-1000). From these and the previous sample points, all sample points that have lower function values than any sample points within a prescribed "critical distance" of themselves are selected as start points for local minimizations. (The critical distance is tied to the statistical properties of the method and decreases at each iteration.) Then local minimizations are conducted from each start point, using a standard unconstrained minimization algorithm such as the BFGS method, and terminating when they find a local minimizer which is generally in the vicinity of the start point. After all the local minimizations are completed, if various stopping conditions have been met, the local minimizer with the lowest function value is selected as the global minimizer. Otherwise, another iteration is performed.

In tests by [6], the stochastic approach has been shown to have attractive computational properties in comparison to other approaches. It also has strong theoretical properties : as the number of iterations and sample points approaches infinity, with probability one the global minimizer is found and the number of local minimizations is finite. For these reasons, we have chosen to base our parallel algorithms upon this approach.

Our static, synchronous parallel version of this algorithm [2] divides the domain space into P equal subregions, where P is the number of available processors. Each processor independently conducts sampling in the subregion it has been assigned, and then selects *candidate start points* from among its sample points, using the same procedure described above but considering only the sample points in its subregion. The processors then synchronize and check, for any candidate start point within the critical distance of a subregion boundary, whether there is a lower sample point within the critical distance of it in some neighboring subregion. If so, the candidate start point is eliminated as a start point. Local minimizations are performed from the remaining start points; these points may not

be equally distributed among the subregions, so they are collected centrally and then distributed to the processors which perform the local minimizations. If there are more start points than processors, initially one start point is distributed to each processor, and the remaining start points are distributed to processors as they finish their current local minimizations. When all the local minimizations have completed, the processors synchronize again in order to determine if the stopping conditions have been satisfied and if not, the process is repeated.

Experimentation with this parallel algorithm showed that it was fairly effective [2][4] but also illuminated some performance problems with the parallel approach. In particular, two aspects of the algorithm sometimes led to load balancing problems, situations where some processors were idle while waiting for other processors to complete their tasks and reach a common synchronization point. One instance was in the start point selection computation. In the efficient versions of the algorithm, before candidate start points are selected, all points with function values greater than some global threshold (called the "cutoff level") are eliminated from consideration. This can leave widely varying numbers of sample points in different subregions and, since the time for candidate start point selection is at least linear in the number of sample points, cause the time for this phase to vary greatly between processors. This causes some processors to wait for others at the synchronization point at the end of this phase. Secondly, the local minimizations also may have widely variable computation times, and there are a variable number of minimizations conducted. Consequently different processors often had differing amounts of work in the local minimization part of the computation.

Another performance problem was observed that is independent of the use of multiple processors. It is that the static algorithm can be inefficient for problems that have local minimizers unevenly concentrated in the domain space. This is because both the parallel and the sequential versions of the algorithm do an equal amount of sampling in all areas of the domain space, even though there may appear to be a greater potential for finding the global minimizer in some portions of the domain than others, for example because more sample points and local minimizers with low function values have been found in some portions than others. While this problem is common to sequential and parallel versions of the stochastic method, it was the consideration of decomposing the domain into subregions, which is only done in the parallel version, that led us to consider alternatives.

To address these problems, this paper will consider the use of *adaptive* and *asynchronous* methods. *Adaptive* methods are ones that react to the current state of a computation by attempting to focus computation on the parts of the problem where this seems most fruitful. In our case, this will involve adjusting the sizes of subregions, as well as the frequency and density of sampling in different subregions. These modifications will not only decrease the total amount of work necessary to solve the

problem, but will also cause the times required to process different subregions to become more similar. *Asynchronous* algorithms are used for parallel computation and allow each part of the computation to proceed independently of other parts. While it is not usually possible to write parallel programs that don't require any synchronization, the intent is to design algorithms whose parts are as autonomous as possible. This allows elimination of processor idle time, but may require additional coordination, and may lead to some unnecessary computations being performed.

In section 2, we present the framework of an adaptive stochastic global optimization algorithm that can be used for sequential computation or as the basis for a parallel implementation. Several adaptive heuristics that we have investigated are described. In section 3 we describe the asynchronous parallel algorithm. We also discuss how the asynchronous and adaptive methods can be used together. Section 4 discusses the experiments we ran to test the algorithms. We present sequential computation results for the static algorithm and for the adaptive algorithm using several different adaptive heuristics, as well as results for all four possible combinations of the static or adaptive, synchronous or asynchronous algorithms in a parallel implementation on a network of Sun workstations. In Section 5, we conclude and present directions for further research.

2 Adaptive Stochastic Global Optimization Algorithms

2.1 Framework of the Adaptive Algorithm

The goal of our adaptive algorithms is to identify portions of the domain space that appear productive and give them more attention, while diverting attention away from portions that are less fruitful. A convenient way to do this is by dividing the domain space into subregions (as is done in our parallel methods), and then to adjust the sizes and/or the amount and frequency of work that is done is different subregions. Specific techniques we will use to do this include splitting of subregions, adjustment of sampling densities, and delayed scheduling of particular subregions.

The general framework for our adaptive global optimization algorithm is given below. (This framework will also serve as the basis for the other new algorithms described in this paper.) We describe the steps of the algorithm from the point of view of a particular subregion s of the overall domain space, S. At this point, nothing is implied about the order of processing the subregions, which could either be synchronous (each subregion completes iteration k before any begins iteration $k+1$) or asynchronous. Similarly, nothing is implied about when the minimization steps are performed, or their placement onto processors in a parallel implementation. In particular, the minimizations in step 4 are not necessarily performed immediately after, or on the same processor as, steps 1-3. Both

of these topics will be addressed in Section 3.

Algorithm 2.1 – Adaptive Global Optimization Algorithm

For each subregion s of the domain space S, at every iteration, k,

1. **Sampling** : Generate a prescribed number of random sample points within s and calculate their function values.

2. **Start point selection** : Select sample points with low function values in s to be start points, using a procedure that is similar to the static algorithm but which does not involve communication with any other subregions.

3. **Adaptive decisions** : Apply a heuristic procedure to determine the sample size for s for the next iteration, whether s should be split, and whether s should be scheduled or skipped at the next iteration.

4. **Local minimizations** : Perform local minimizations, if any, from the start points generated at step 2.

The difference between the start point selection method used in step 2 and the one used in the static algorithm is relatively small and is motivated largely by the possibility of an asynchronous implementation. It will be discussed in Section 3.3. The remainder of this section discusses some possible ways to make the adaptive decisions at step 3.

2.2 Adaptive heuristics

We have investigated two different measures that indicate how *productive* a subregion of the domain space is, i.e. how likely it is that sampling in this subregion will lead to the discovery of low minimizers and possibly the global minimizer. The first measure, the number of local minimization start points generated in the subregion so far, may be an indicator of how many minimizers remain to be found in that subregion. This measure has not proven to be especially effective in identifying subregions that are productive, and will not be considered further. The second measure is the percentage of sample points in the subregion whose function value falls below some global threshold. This threshold, called the *cutoff level*, is calculated at the first iteration of most stochastic methods, and is defined as the function value for which X% of the sample points (say 20%) have lower values and the remaining

(100-X)% have higher values. The percentage of sample points in a particular subregion whose function value is below the cutoff level can vary greatly, and has proven to be effective in identifying productive subregions. Hence, the heuristics discussed in the remainder of this section will be based on this measure of productivity.

Once we have a method for identifying productive regions, we need to determine how to use this information to devote more attention to productive subregions and divert attention away from less productive subregions. We have investigated three techniques, that we use together, for accomplishing this. *Splitting* a productive subregion into two smaller subregions is used to devote more attention to that portion of the domain. The intent is to have the density (number divided by volume of subregion) of the sample points that are generated at each iteration be greater in productive subregions than in less productive regions. Splitting allows this to be done while keeping the cost of processing each subregion nearly constant. In fact, as we will see in Section 4, it may reduce the total cost of start point selection since this cost is generally more than linear in the number of sample points below the cutoff level. In parallel implementations it also helps load balancing and allows more of the processors to work on productive portions of the domain space. *Sample size reduction* and *postponement* techniques have been used to divert attention away from subregions that are not productive. Sample size reduction reduces the density of sample points generated at each iteration in unproductive subregions. Postponement suspends processing of an unproductive subregion for one or more iterations.

Below we describe formally how we have implemented these techniques. An interesting feature of these heuristics is that we never increase the density of the sample points generated per iteration, instead it is kept the same for productive subregions and decreased for the remainder of the subregions. This was found to be more efficient than increasing the density for productive subregions. First we need some definitions.

Definitions

C - The global cutoff level.

P_{below} - The percentage of sample points in some subregion whose function values fall *below* the cutoff level C.

SPLIT-P - The percentage above which a subregion is considered to be productive.

S_i - The ith subregion of the domain space S.

$sample\text{-}size_i$ - The total sample size of S_i (over all iterations).

sample-size-incr$_i$ - The number of new sample points generated in S_i at each iteration.

SPLIT-SIZE - The lowest total sample size at which a subregion is allowed to split.

REDUCTION - Factor used for sample size reduction.

MIN-SAMPLE-INCR - The minimum sample size increment that a subregion must maintain.

POSTPONE(s, DELAY) - Postpone the scheduling of subregion s by *DELAY* iterations.

1. **Subregion splitting**

   As in all the heuristics, the percentage of sample points whose function value is below the global cutoff level is used to determine if a subregion is productive. If it is, the density of new sample points is kept the same. The subregion is split into two smaller subregions, unless the total sample size already is below some threshold, in which case the subregion is not split.

   Heuristic: If $P_{below} > SPLIT\text{-}P$ and $sample\text{-}size_i > SPLIT\text{-}SIZE$
   then
   S_i is split into S_{i0} and S_{i1}
   $sample\text{-}size\text{-}incr_{i0}, sample\text{-}size\text{-}incr_{i1} \leftarrow sample\text{-}size\text{-}incr_i/2$
   else
   S_i, $sample\text{-}size\text{-}incr_i$ remain the same

2. **Sample size increment reduction**

   If the subregion is not productive, this heuristic reduces the sample size increment, unless it has already reached some minimum value in which case it is not changed.

   Heuristic: If $P_{below} < SPLIT\text{-}P$
   then
   if $sample\text{-}size\text{-}incr_i > MIN\text{-}SAMPLE\text{-}INCR$
   then $sample\text{-}size\text{-}incr_i \leftarrow sample\text{-}size\text{-}incr_i/REDUCTION$
   else
   $sample\text{-}size\text{-}incr_i$ remains the same

3. **Subregion splitting and postponement with sample size reduction**

   This heuristic combines the splitting and sample size increment reduction heuristics described above. In addition, if the sample size increment of an unproductive subregion is already at or below the minimum value, then the processing of this subregion is suspended for one or more iterations.

```
Heuristic:  If P_below < SPLIT-P
              then
                 if sample-size-incr_i > MIN-SAMPLE-INCR
                   then sample-size-incr_i ← sample-size-incr_i/REDUCTION
                   else POSTPONE(S_i, DELAY)
              else
                 if sample-size_i > SPLIT-SIZE
                   then
                      S_i is split into S_i0 and S_i1
                      if sample-size-incr_i > MIN-SAMPLE-INCR
                        then sample-size-incr_i ← sample-size-incr_i/2
                        else sample-size-incr_i remains the same
                   else
                      S_i remains the same
                      sample-size-incr_i remains the same
```

In section 4, we will discuss experiments that were performed using each of the adaptive heuristics. The third heuristic will be seen to give the best improvements over the static algorithm. The main reason is that it both focuses attention on productive subregions and diverts attention away from less productive subregions.

3 An asynchronous, stochastic parallel global optimization algorithm

In this section we present an asynchronous, stochastic parallel global optimization algorithm and describe its implementation on a distributed memory computer. We also discuss how the asynchronous parallel approach can incorporate adaptive techniques.

Recall from section 1 that the static, synchronous, stochastic parallel global optimization algorithm is an iterative method with the following steps. At iteration k:

1. Generate sample points in all the subregions and calculate their function values.

2. Select candidate start points for local minimizations from all the subregions.

3. Wait for all processors to finish their start point selection phase, and then eliminate candidate start points for which there are lower sample points within the critical distance in neighboring subregions.

4. Perform local minimizations from all start points generated at steps 2-3 by distributing one minimization at a time to each processor.

5. Wait for all minimizations to complete, then decide whether or not to stop.

As we mentioned in Section 1.1, some processors can have much more work than others in steps 2 and 4. Because of this, some processors may be waiting for a long time, in an idle state, for the synchronizations at steps 3 and 5. In step 2, the imbalance in workload arises because the time to perform the start point selection is dependent upon the number of sample points in the subregion below the global cutoff level, which can vary greatly, even though the total number of sample points in each subregion is the same in the static algorithm. In step 4, the imbalance arises because the number of iterations and function evaluations required for a local minimization may be highly variable and cannot be predicted a priori. In addition, the number of local minimizations may not be a multiple of, or even as large as, the number of processors.

The asynchronous algorithm addresses this load imbalance by eliminating the synchronization at steps 3 and 5. To do this it changes the overall viewpoint of the method from the global one used above to the subregion-based perspective already used in Section 2. The basic tasks are the same, in particular sampling/start point selection, and local minimizations. But the precedence relations between these tasks are relaxed or removed, and the order that they are conducted in may change.

To implement the asynchronous algorithm, there must be some mechanism for controlling the entire process, i.e. for scheduling the tasks and assigning them to processors. In conjunction, there must be a mechanism for stopping the entire algorithm. In addition, the start point selection algorithm needs to be modified, so that it doesn't require the task performed at a global level in step 3 above, but still eliminates unnecessary start points efficiently. The remainder of this section discusses these three issues, as well as the incorporation of adaptive techniques into the asynchronous method, and the overall advantages and disadvantages of the asynchronous approach.

3.1 A centralized control implementation

We have used a *centralized control* organization to implement the asynchronous parallel algorithm in a distributed memory computing environment, which is very similar to the organization of Schedule [3], a system developed by Dongarra and Sorenson for centralized scheduling of FORTRAN programs in a shared memory environment. By centralized control, we mean that there is a set of master processes that are responsible for scheduling and placement of tasks and manipulation of global data structures, and that control a set of slave processes. The slave processes perform the computational tasks of the asynchronous algorithm, which we refer to as *schedulable tasks*. The two schedulable tasks of this algorithm are *subregion* tasks and *local minimization* tasks. Subregion tasks refer to

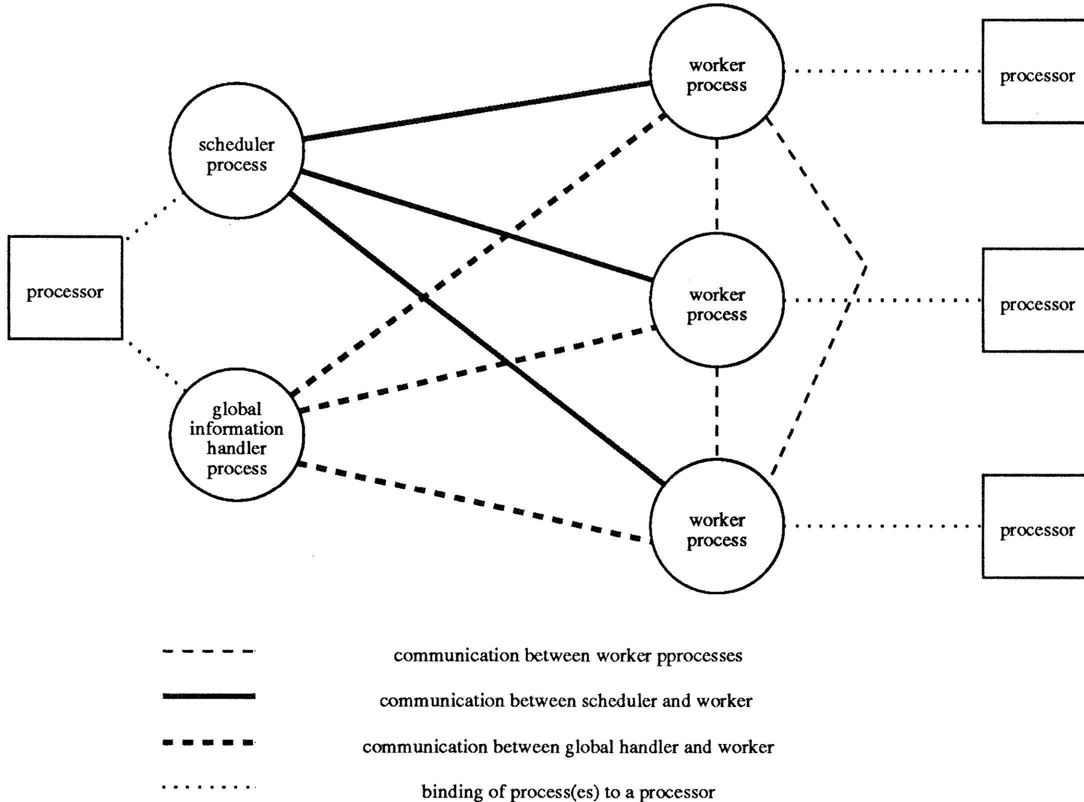


Figure 1: Communication and processor bindings

one iteration of sampling and start point selection for a given subregion (steps 1-2 of Alg. 2.1, including the modification discussed in section 3.3), and local minimization tasks refer to a single local minimization from step 4 above.

Our implementation of the asynchronous algorithm uses three types of processes, whose communication and processor bindings are depicted in figure 1. The *scheduler* process is the primary master process and provides the centralized control of the computation, determining where and when schedulable tasks should execute, gathering results from these tasks, and deciding when the computation should terminate. *Worker* processes are the slave processes of the computation. Each worker process exists for the duration of the computation and is statically bound to a processor. It is continually assigned schedulable tasks, either subregion or local minimization, by the scheduler, and also communicates with other worker processes. The *global information handler* process is the secondary master process. It manages a data structure that keeps track of all the minimizers that have been found, and the oversample points that have been generated (see Section 3.3). Worker processes request this sample information from the global information handler when they are performing a subregion task. Generally the scheduler and the global information handler are statically bound to the same

processor.

A high level pseudo code description of each type of process is given in figures 2, 3, and 4. These descriptions provide the basic description of our asynchronous algorithm, but do not specify two important details, namely how schedulable tasks are prioritized, and how they are mapped to processors. These issues are discussed next.

The scheduler determines priorities among the currently schedulable tasks as follows. First, schedulable tasks from a lower numbered iteration have higher priority than tasks from a higher numbered iteration. Secondly, within the same iteration, subregion tasks have higher priority than local minimization tasks. Finally, schedulable subregion tasks, or local minimization tasks, from the same iteration are scheduled first come first served, with the one possible exception mentioned in the next paragraph.

Any subregion and local minimization tasks can be placed onto any available processor. There is no attempt (or need) to try to place specific tasks onto specific processors, with the following exception. If there are multiple subregion tasks that share the highest priority, and/or multiple free processors, then, if possible, subregions are assigned to the same processor where they were processed at their last iteration. This is done to avoid the movement of sample points from one processor to another. It is often not possible, however, due to the asynchronous nature of the algorithm.

3.2 Termination of the asynchronous algorithm

As shown in Figure 2, when the scheduler process determines that an iteration is complete, it checks the termination criteria. Iteration k is complete if iteration $k-1$ is complete, all subregion tasks for iteration k have been completed, and all local minimization tasks generated by subregion tasks from iteration k have been completed. Stopping criteria similar to those in [6], which are derived from the probabilistic analysis of the method, could be used. In our tests, however, we have simply stopped when a given percentage of the (known) number of local minimizers was found, because we found that this led to a fairer comparison of the methods.

At the time when the algorithm is terminated, some processing of subregion and possibly local minimization tasks from subsequent iterations will already have been conducted. This "runover" work is unnecessary, and is one of the main tradeoffs that one must accept in return for the improvement in load balancing that comes from eliminating synchronization.

```
Initialize task queue with subregion tasks for initial subregions
REPEAT
    if a processor is available
       then schedule a task with the highest priority
    generate new tasks from completed tasks and insert in priority order
       - create new local minimization task for each
            new local minimization start point
       - create new subregion task(s) for iteration k+1 (or higher)
            for each subregion task that has completed iteration k
    if all steps of an iteration have completed
       then decide whether or not to stop
UNTIL stopping criteria have been satisfied
```

Figure 2: *scheduler* process

```
REPEAT
    wait for a new task
    if new task = local minimization
       then find a local minimum starting from the given start point
    else {new task = subregion task}
       generate new sample points
           - acquire previous sample points for this subregion
                from other processors
           - acquire minimizers and oversample points from gl. info. handler
       select new start points
       determine new sample size increment for this subregion and whether to
            split the subregion; inform scheduler and global info. handler
       send new start points to scheduler
    request new work
UNTIL scheduler stops me
```

Figure 3: a *worker* process

```
Initialize data structures
REPEAT
    receive message
    if message = send oversample and minimizer information
       then send information to the appropriate worker process
    if message = split request
       then update data structure to reflect subregion split
    if message = add oversample points or minimizers
       then update data structure to add new points
UNTIL scheduler stops me
```

Figure 4: *global information handler* process

3.3 Start point selection in the asynchronous parallel algorithm

The sequential stochastic method of [6] selects start points for local minimizations as follows. At each iteration, a critical distance is calculated which is a function of the largest possible sample density (i.e. no consideration is made for sample size increment reduction) and the iteration, and is tied to the probabilistic analysis of the method. Sample points that have the lowest function values among all sample points within the critical distance from themselves are selected as start points.

Our static, synchronous parallel algorithm achieves the same result in a slightly different manner. First *candidate start points* are selected that have the lowest function values among all sample points *in that subregion* within the critical distance from themselves. Then candidate start points that are within the critical distance of another subregion(s) are compared to sample points from that subregion(s), and are eliminated as start points if a lower sample point is found within the critical distance.

Our asynchronous algorithm modifies this procedure in the following way. Candidate start points for a given subregion are still selected as described above for the synchronous algorithm, but after this is done, it is no longer feasible to compare candidate start points near subregion boundaries to current sample points from neighboring subregions. The reasons why this is infeasible include the lack of synchronization, the migration of sample point data between processors from iteration to iteration, and the possible adaptive splitting of subregions. Instead, a technique called *oversampling* is used. Suppose that the subregion's sample density is D, that the critical sphere (sphere with radius equal to the critical distance) around some candidate start point falls partly outside the subregion, and that the volume of this sphere is V. At most $D * V$ new random sample points, called *oversample points*, are generated within this sphere. As each oversample point is generated, it is checked to see whether it falls outside the subregion, and if so its function value is calculated. (If none of the oversample points falls outside the subregion, then one oversample point that is within the critical sphere, but outside the subregion, is generated in a non-random manner.) If any oversample point has a lower function value than the candidate start point's, then the candidate start point is eliminated, otherwise the candidate start point is used as a local minimization start point.

The extra cost of this oversampling procedure is fairly small, and is another tradeoff for the improvement in load balancing that comes from eliminating synchronization. Function evaluations at oversample points are utilized further by shipping them and their values to the subregion they reside in (see Figures 3 and 4).

3.4 Incorporation of adaptive features, and implementation considerations

The adaptive features discussed in Section 2 fit naturally into the asynchronous global optimization algorithm described in Figures 2-4 above. In fact, the high-level description in Figures 2-4 already allows for adaptive features, in the third step of the subregion task portion of the worker process and the second message type of the global information handler. The combination of the adaptive and asynchronous approaches is natural because both are based on an individual subregion-oriented view of the problem, rather than an iteration-based view, or a view based on the overall domain space.

To implement an adaptive, asynchronous algorithm we use dynamic data structures in each of the three types of processes. In the scheduler process, subregions are represented as tasks that are present either in a ready queue, if they are not currently executing, or an active queue, if they are executing. The task queues allow a dynamic number of subregion and local minimization tasks. When a subregion splits, the scheduler creates two new tasks that reflect the new subregion boundaries, their iteration, and other information. Subregion tasks that are postponed for execution remain in the ready queue until some future iteration. At the worker process, sample points are included as part of a subregion, so that when a subregion splits, the sample is also split between two subregions. Finally, the global information handler keeps track of the oversample points for each subregion. When a subregion is split, the global information handler updates a tree that has as its leaves the current subregions and their oversample points.

3.5 Advantages and disadvantages of the asynchronous approach

The goal of the asynchronous approach was to avoid the idle processor times that occur in the synchronous global optimization algorithm due to the variability in the computational times of subregion tasks, or local minimization tasks, in a given phase. By allowing the overlapping of tasks from different phases or iterations, the asynchronous algorithm should accomplish this goal. Idle times should only occur due to communication delays, or if there are not as many schedulable tasks as processors. The latter situation is easily avoided by using enough subregions.

The discussion in this section should also make it apparent that there are several new costs inherent in the asynchronous approach. First, there is the additional cost of the master processes (the scheduler and global information handler) as compared to the relatively small cost of the master process for the synchronous algorithm. Secondly, as mentioned in section 3.2, the asynchronous algorithm will suffer from "runover" costs, because by the time it decides to terminate the entire algorithm after some iteration k is complete, some unnecessary work at higher iterations will have

Figure 5: 46 minimizer problem

been performed. Also, the asynchronous start point selection procedure described in Section 3.3 requires some extra function evaluations at the oversample points and may still result in more start points than in the synchronous algorithm, which would require extra local minimizations. Finally, more data movement is required in the asynchronous algorithm, since subregions do not necessarily execute on the same processor at different iterations and if not, the existing sample points must be shipped between two processors. The computational results in Section 4 will shed some light on the effects of these factors.

4 Computational Results

In this section, we present experimental results of tests we have run on the global optimization algorithms discussed in Sections 2 and 3. First we discuss sequential tests comparing the static algorithm and the three adaptive heuristics. Next, we discuss parallel tests comparing all four possible combinations of static and adaptive, synchronous and asynchronous algorithms.

4.1 Construction of the experiments

4.1.1 Test problems

All the experiments reported in this section use a single artificial test problem that we constructed. The reason we constructed a new problem is that there are very few global optimization test problems, and they all seemed unsatisfactory for testing these algorithms, either because they had very few minimizers (e.g. the set used by [6]) or too many minimizers to be affordable for the testing we performed (e.g. the set used by [5]).

Our test function is the two variable function depicted in Figure 5. Both x_1 and x_2 range between 0 and 8. Each box illustrated contains the fourth order polynomial

$$f(x) = (box\ number) + (x_1 - c_1)^2 + (x_2 - c_2)^2 + (1 + 2 * (x_1 - c_1)^2 + 3 * (x_2 - c_2)^2)^2$$

where c_1 and c_2 are the coordinates of the center of this box. That is, there is a local minimizer at the center of each box, whose function value is equal to the box number. (The overall function is discontinuous at the borders between boxes, but this is unimportant to the algorithms discussed in this paper.) Thus the test problem has an uneven concentration of minimizers, with the global minimizer having a relatively small region of attraction. Such a function should give an indication of the best type of improvements possible from the algorithms described in this paper. We feel that examining our algorithm under such conditions is useful for understanding whether the approach has any utility, and differentiating between variations upon it.

The methods under consideration perform differently depending on the initial sample size, and the initial number of subregions. Thus, for each experiment, we ran tests using several different combinations of these initial parameters. For each combination, we performed 10 runs and averaged the results. This was necessary because of the stochastic nature of the algorithms. Also, to make our testing more representative of real problems, where function evaluation is usually moderately to very expensive, we have added irrelevant computations to the function evaluation so that its cost is about 0.2 seconds on a Sun 3/60.

Due to the number of different configurations tested, the need to perform 10 runs per configuration, and the considerable cost of each run, the tests reported herein are very expensive and time consuming. For this reason, we have not yet been able to try other test problems, but intend to involve a more complete set of test problems in future tests.

4.1.2 Test algorithms

We tested static and adaptive, synchronous and asynchronous versions of the methods described in Sections 2 and 3, in both sequential and parallel environments. For convenience, all the test algorithms used the framework outlined in Figures 2-4. This framework is intended to support the asynchronous algorithm, and uses three types of processes (scheduler, global handler, and workers). To construct synchronous algorithms using this framework, we simply modify the priorities so that no iteration $k+1$ subregion process can execute before iteration k is complete. To construct sequential algorithms, we use one worker process. In the static, synchronous case, the overhead of this implementation is a little higher than for an algorithm constructed specifically for this case, but the differences are not significant. In the adaptive, synchronous case, most of this framework is necessary anyhow. In the sequential cases, it might have been slightly cheaper to run a one-process algorithm (though the program would be more difficult to write), but again the differences are not significant and are mitigated by the factors discussed below.

4.1.3 Testing environment

The sequential and parallel experiments were run on a network of Sun 3/60 workstations. We first ran several of the sequential experiments on a single workstation, meaning that the scheduler process, the global handler process, and a single worker process were all on one processor. This proved to be extremely time consuming because most of the time was spent in context switching between the three processes. As a result, the overall elapsed time of the computation did not provide a reasonable basis for evaluating the speed of the algorithm. Instead, we found that if we considered the separate times for each of the three processes, the worker process time was by far the largest and provided a good measure of the time really required by the algorithm. We also found that if we placed the scheduler process and global information handler process on one workstation, and the worker process on another, then the overall elapsed computation time came within 2 % of the the worker process time in the one-processor experiments. These two processor experiments do not suffer from the problems of context switching, are much faster, and are able to provide us with a reasonable indication of the sequential computation speed. Consequently, all of the sequential experiments that we report in this section are performed in this manner. Note that this method of calculating sequential times underestimates them slightly, so that speedups based upon them also are slight underestimates.

The parallel experiments were run on a network of nine Sun workstations, with eight workstations dedicated to worker processes and one workstation dedicated to both the scheduler process and global

information handler process.

4.1.4 Reporting of Test Results

For each configuration of algorithm, initial parameters, and number of processors that we tested, we report the averages (over the 10 runs) of the number of complete iterations performed by the algorithm, the total number of function evaluations performed, the total number of sample points generated, the total number of local searches performed, and the total computation time. The number of function evaluations includes one at each sample point, plus all those performed in the local minimizations (including n per finite difference gradient evaluation). The computation times are the actual elapsed times required by the algorithm.

4.2 Sequential experiments

We examined the performance of the sequential, synchronous global optimization algorithm using the static method and each of the three adaptive heuristics described in Section 2.2. Recall that the first adaptive heuristic is *subregion splitting*, the second is *sample size increment reduction*, and the third is *subregion splitting and postponement with sample size increment reduction.* The experimental results are summarized in Table 1.

In experimenting with the adaptive methods, it became apparent that neither of the first two heuristics was, by itself, sufficient to consistently realize a significant improvement over the static algorithm. Since each of these heuristics addresses a different problem, the best results were obtained by combining the two heuristics, and adding the postponement technique, as is done in the third heuristic. We found that this heuristic yielded considerable improvements over the static synchronous algorithm.

The following discussion analyzes the performance of each of the adaptive heuristics.

1. **Subregion splitting**

   The cutoff level measure is a good way of identifying subregions that are productive. However, splitting alone does not effectively concentrate the computation in the more productive subregions, because it does not adjust the sample densities. Instead, splitting has two other benefits for sequential computation. First, it significantly reduces the time for start point selection. This is because the productive subregions have the most sample points below the cutoff level, and the time required for start point selection is more than linear in this number of points. Thus splitting the productive subregions reduces the time for computing start points in the

initial sample = 1000, 8 subregions	Static Synch	Heuristic 1	Heuristic 2	Heuristic 3
total number of iterations	8.6	6	8.2	6.1
total sample size	8689	6121	5110	3844
total number of local searches	44	54	43	45
total function evals	9624	7277	5110	4759
fnc eval improvement over static case	-	24 %	47 %	50.5 %
computation time	84:03	31:22	68:20	23:21
time improvement over static case	-	63 %	19 %	72 %
initial sample = 1000, 16 subregions	Static Synch	Heuristic 1	Heuristic 2	Heuristic 3
total number of iterations	6	5.2	5.8	5.2
total sample size	6263	5534	3880	3095
total number of local searches	68	76.8	67	51
total function evals	7666	7102	5270	4109
fnc eval improvement over static case	-	7 %	31 %	46 %
computation time	37:49	30:36	29:38	20:47
time improvement over static case	-	19 %	22 %	45 %

Table 1: Performance statistics for the adaptive heuristics - sequential tests

subregions where this is most expensive. Second, we observed that there tend to be more local searches per iteration when subregions are split, and that they lead to the discovery of more local minimizers, and hence the termination of the algorithm, in fewer iterations. This is because when the subregion is smaller, there are more candidate start points, and empirically it seems that the oversampling procedure of Section 3.3 does not eliminate as many of these as would be eliminated if the subregion were bigger. The earlier, extra searches turn out to be beneficial.

The combination of these effects account for the substantial decreases in computation time (19 % and 63 %), and the smaller decreases in iterations, total sample size, and function evaluations for heuristic 1 shown in Table 1. Note that another possible benefit from splitting, allowing more processors to concentrate on productive subregions, is not relevant for sequential computation.

2. **Sample size reduction**

   The sample densities of unproductive subregions are reduced by this heuristic, which is reflected in the decrease in total sample size required to solve the problems. This decrease is not as large as it could be because unproductive subregions tend to reach the lower limit for sample size increment and cannot be reduced further. (This observation led us to consider using postponement as is done in heuristic 3.) The benefits from splitting, namely reduction in computation time required for start point selection, and more searches per iteration leading to fewer total iterations, are not achieved.

3. **Subregion splitting and postponement with sample size reduction**
   This heuristic effectively combines the benefits of the previous two. The cutoff level identifies productive subregions well. Sample size reduction combined with postponement effectively serve to concentrate most of the sampling and searching in the productive subregions. Splitting serves to reduce the computation time of start point selection in the areas where it is expensive, and to generate more start points per iteration, which allows the computation to terminate in fewer iterations. The overall result is a substantial reduction in both computation time (45 % and 72 %) and the total number of sample points and function evaluations (about 50 %).

4.3 Parallel experiments

In this section we present our computational results for the four parallel algorithms, namely the static, synchronous algorithm, the static, asynchronous algorithm, the adaptive, synchronous algorithm, and the adaptive, asynchronous algorithm. In both adaptive algorithms, the adaptive heuristic used is subregion splitting and postponement with sample size reduction. In comparing the algorithms, we will attempt to isolate how the adaptive aspect and the asynchronous aspect each contribute to the improvement over the static synchronous case. We also discuss the speedups of the parallel algorithms over their sequential counterparts.

4.3.1 Comparison of the four parallel algorithms

The results of the four parallel algorithms are summarized in Table 2. Each algorithm has been tested using five different initial configurations for sample size and number of subregions.

The results show that the asynchronous algorithms are moderately effective in reducing the total computation time of their synchronous counterparts. In the static cases with 8 subregions, the reductions by the asynchronous algorithm are fairly large (about 35 % in 2 of the 3 cases). This is because both the synchronous and the asynchronous algorithms have fairly large, but irregular sized tasks, and so the asynchronous approach leads to much better load balancing. The improvements by the asynchronous algorithms are also good for the adaptive cases with 8 initial subregions (up to 35 %), even though the adaptive approach could be expected to itself improve load balancing and thus diminish the potential for further improvements. In both the static and adaptive cases with 16 initial subregions, however, the times required by the synchronous and asynchronous methods are about the same. This is apparently because the load balancing of the synchronous algorithm is already rather good due to the larger number of subregions and smaller task sizes, so that the improvement in load

density = 500, 8 subregions	Static Synch	Static Asynch	Adapt Synch	Adapt Asynch
total number of iterations	13.9	11.1	11.5	9.4
total sample density	6977	6319	3591	2784
total number of local searches	40.4	50	58	63.5
total function evals	7816	7383	4808	4133
fnc eval improvement over static synch	-	5 %	38 %	47 %
computation time	23:06	14:26	5:00	3:16
time improvement over static synch	-	37 %	78 %	86 %

density = 1000, 8 subregions	Static Synch	Static Asynch	Adapt Synch	Adapt Asynch
total number of iterations	8.6	7	6.5	6.1
total sample density	8383	8532	4026	3963
total number of local searches	39.6	46	47	53
total function evals	9186	9474	4966	5055
fnc eval improvement over static synch	-	-3 %	46 %	45 %
computation time	23:19	15:01	4:10	3:25
time improvement over static synch	-	35 %	82 %	85 %

density = 2000, 8 subregions	Static Synch	Static Asynch	Adapt Synch	Adapt Asynch
total number of iterations	4	4	3.8	3.9
total sample density	8077	10592	5591	5850
total number of local searches	36.4	87	48	55
total function evals	8734	11391	6548	6926
fnc eval improvement over static synch	-	-23 %	25 %	21 %
computation time	14:22	13:30	4:29	4:05
time improvement over static synch	-	6 %	69 %	71 %

density = 1000, 16 subregions	Static Synch	Static Asynch	Adapt Synch	Adapt Asynch
total number of iterations	5.6	5.7	5.4	5.7
total sample size	5891	5010	3048	3369
total number of local searches	69.9	81.5	45	69
total function evals	7341	6697	3976	4788
fnc eval improvement over static synch	-	9 %	46 %	35 %
computation time	5:04	5:10	3:33	3:23
time improvement over static synch	-	-2 %	30 %	33 %

density = 2000, 16 subregions	Static Synch	Static Asynch	Adapt Synch	Adapt Asynch
total number of iterations	3.6	3.6	3.4	3.5
total sample density	7488	7958	5261	5152
total number of local searches	51.9	66	62	60
total function evals	8510	9266	6529	6326
fnc eval improvement over static synch	-	-9 %	23 %	26 %
computation time	5:51	5:48	3:54	3:39
time improvement over static synch	-	-0.8 %	33 %	37 %

Table 2: Performance comparison of the static synchronous, static asynchronous, adaptive synchronous and adaptive asynchronous algorithms

balancing from the asynchronous approach is offset by the additional work for extra partial iterations (see Section 3.2).

The improvements in going from static to adaptive algorithms are quite impressive in all cases. In the synchronous cases the reductions in computation time range from 30 % to 82 %, while in the asynchronous cases they range from 34 % to 77 %. These results are consistent with the sequential results in Section 4.2; indeed we see that the improvements by the adaptive approach sometimes are even larger in the parallel case, presumably because, in addition to the advantages discussed in Section 4.2, the adaptive techniques also improve load balancing.

The overall reductions in computation time between the static synchronous and the adaptive asynchronous algorithm range from 33 % to 86 %. In spite of these improvements, however, there are two points that cannot be overlooked. First, the number of local searches for the adaptive and asynchronous methods is always higher than for the static, synchronous method. One reason seems to be that the adaptive methods have more subregions, which lead to more candidate start points, and the oversampling phase of the algorithm is not entirely effective at eliminating the sample points that will lead to redundant searches. We intend to investigate ways to reduce this problem in the future, either by changing the method for selecting start points, or by halting searches that appear to be redundant prematurely. A second reason is apparently the "runover" effect of the asynchronous algorithm, i.e. that extra searches are performed at iterations beyond the eventual stopping point. This effect seems to significantly increase the number of searches between the synchronous and asynchronous static algorithms, but not between the synchronous and asynchronous adaptive methods. Since the adaptive methods appear preferable, this effect is not a major concern.

Secondly, it must be noted that the static synchronous algorithm is far more efficient with a higher number of initial subregions than it is with a smaller number of initial subregions. Consequently, the improvements by the adaptive, asynchronous algorithm are not as dramatic in this case (although they are still around 33 %). The reason that the synchronous algorithm is more efficient with more subregions is that the cost of start point selection, which is a significant portion of the total time, decreases significantly as the number of subregions are increased. This also improves load balancing which further decreases the execution time. For complicated functions with more variables, however, an inordinately large number of initial subregions might be necessary to make the start point selection times small for the static algorithm, but the overhead costs associated with this number of subregions might quickly become prohibitive. In these situations, adaptive methods should be very useful.

Finally, note that the improvement in the total number of function evaluations required by the *best* adaptive, asynchronous method (best in terms of function evaluations) over the *best* static,

	Speedup 1	Speedup 2	Speedup 3	Speedup 4
density = 1000 8 subregions	3.6	5.6	5.6	6.8
density = 1000 16 subregions	7.4	7.3	5.85	6.1

Table 3: Speedups of the parallel algorithms

synchronous method is 47 %. Thus the adaptive, asynchronous method would also be advantageous if function evaluation was the dominant cost. Interestingly, the smallest total number of function evaluations for the adaptive, asynchronous method comes from using the smallest initial sample size and number of subregions. The time is also lowest in this case, although the variation in computation time for the adaptive, asynchronous algorithm between the different starting configurations is small, whereas the variation in total function evaluations is significant.

4.3.2 Speedups

Table 3 gives the speedups, in computation time, by the parallel algorithms over their sequential counterparts, for the two sets of experimental configurations (initial sample size and number of subregions) that are common to Tables 1 and 2. The 4 columns of Table 3 are the following:

1. **Speedup 1** - *Static Synchronous Sequential/Static Synchronous Parallel*
2. **Speedup 2** - *Static Synchronous Sequential/Static Asynchronous Parallel*
3. **Speedup 3** - *Adaptive Synchronous Sequential/Adaptive Synchronous Parallel*
4. **Speedup 4** - *Adaptive Synchronous Sequential/Adaptive Asynchronous Parallel*

In the sequential experiments we used one worker process (on one processor), and in the parallel experiments we used 8 worker processes (on 8 processors). Thus we consider the logical parallelism of the experiments to be 8, even though we used an additional processor for the scheduler and global information handler processes in all the experiments.

For the static experiments, notice that the speedup in the synchronous 8 subregion case (speedup 1) is low, due to the load imbalance associated with using a small number of subregions. The speedup improves considerably by switching to an asynchronous parallel algorithm (speedup 2). In the 16 subregion case, the load balance is much better, and the speedups for both the synchronous and asynchronous algorithms are over 7.

For the adaptive experiments, the speedups in the synchronous cases (speedup 3) are reasonably good, but there still is evidence of load imbalance with both 8 and 16 subregions. The asynchronous cases (speedup 4) show good speedups, but since the adaptive sequential algorithms are far more efficient than the static sequential algorithms, the overhead of their parallel counterparts now becomes more significant and prevents the speedups from being higher than 6-7. This overhead consists of extra local searches, due to less efficient elimination of redundant start points, and extra sampling, due to subregions that perform work at an iteration beyond the one where the asynchronous algorithm actually terminates.

5 Summary and Future Directions

We have presented a family of algorithms for stochastic global optimization that use adaptive and asynchronous techniques. In our tests, these techniques have improved the speed of static and synchronous algorithms considerably, on both sequential and parallel computers. For sequential algorithms, an adaptive heuristic that uses subregion splitting, sample size increment reduction, and postponement, is up to 72 % faster than the static sequential global optimization algorithm. For parallel algorithms, we have demonstrated that a static parallel algorithm may be improved solely by using asynchronous techniques, and that an adaptive asynchronous parallel algorithm is up to 86 % faster than the static synchronous parallel algorithm.

Our research has also demonstrated several shortcomings of our new methods, however, and part of our ongoing research is to investigate ways to improve the algorithms. In addition, we are interested in experimenting with alternative implementations of the parallel algorithms, such as a parallel asynchronous implementation that does not have a central scheduler process. In the remainder of this section, we briefly discuss some of the interesting directions for future research.

Before we take steps to improve the parallel algorithms, we intend to do some more extensive tests that will further isolate the factors contributing to improvements in performance, and the factors that are causing extra work to be done. For example, in the paper we have hypothesized that the oversampling technique (Section 3.3) contributes to the large number of local searches that are observed in the new parallel algorithms, and that this could be viewed as an unnecessary extra cost. On the other hand, we also observed that the algorithms that run faster do so because the extra local searches cause them to require fewer iterations. Thus the question remains whether oversampling is creating unnecessary searches, or whether these "extra" searches are indeed necessary to speed up the solution of the problem. To investigate this question, we will run more tests, with and without

oversampling. Our experimentation will also include running test problems of higher dimensions and with a larger number of local minimizers. We suspect that the differences between the performance of adaptive and static algorithms will be greater with these larger problems.

Another possibility for reducing the cost of extra local searches in our new methods is to incorporate heuristics into the local minimization procedure that terminate redundant minimizations prematurely. If the local minimization procedure has some knowledge about minimizers previously encountered, it can discontinue minimizations when they begin to search in the vicinity of a known minimizer. This might eliminate a large portion of the cost of the unnecessary local searches.

In order to improve the parallel algorithms, we can take steps to reduce some of the overhead costs that arise from asynchronous "runover" (see Section 3.2). "Runover" costs can be reduced by simply going back to a synchronous algorithm, but our results show that the adaptive asynchronous parallel algorithm consistently has the best performance. A question that remains is whether we can make the performance of the adaptive synchronous algorithm be closer to that of the adaptive asynchronous algorithm. One possibility is to fine tune the splitting heuristics so that subregion tasks are as regular as possible, thus reducing load imbalance.

Finally, we are concerned with the parallel computation issues that the adaptive and asynchronous algorithms present. One of the important issues surrounding the design and implementation of parallel algorithms is what type of algorithm is best suited for a particular parallel architecture. The algorithms presented in this paper use centralized control for scheduling tasks, terminating the algorithm, or accessing global data. An alternative implementation that we are exploring uses distributed control. A distributed control algorithm does not have a centralized mechanism for scheduling tasks, distributing work, determining when a computation should terminate, or accessing global data, instead, each of these issues is handled locally. We are currently evaluating different possible distributed control implementations using simulation; among the issues that are interesting are different ways to distribute work and determine termination. We will ultimately design and implement a distributed control version of a parallel global optimization algorithm to run on a parallel computer.

References

[1] F. Aluffi-Pentini, V. Parisi, and F. Zirilli. A global optimization algorithm using stochastic differential equations. *ACM Transactions on Mathematical Software*, 14:345–365, 1988.

[2] R. H. Byrd, C. L. Dert, A. H. G. Rinnooy Kan, and R. B. Schnabel. *Concurrent Stochastic Methods for Global Optimization.* Technical Report CU-CS-338-86, Department of Computer

Science - University of Colorado, Boulder, June 1986. To appear in *Mathematical Programming.*

[3] J. J. Dongarra and D. C. Sorenson. Schedule: tools for developing and analyzing parallel fortran programs. In D. B. Gannon, L.H. Jamieson, and R. J. Douglass, editors, *Characteristics of Parallel Algorithms*, pages 363 – 394, MIT Press, 1987.

[4] E. Eskow and R. B. Schnabel. *Mathematical Modeling of a Parallel Global Optimization Algorithm.* Technical Report CU-CS-395-88, Department of Computer Science - University of Colorado, Boulder, April 1988. To appear in *Parallel Computing.*

[5] A.V. Levy and A. Montalvo. The tunneling algorithms for the global minimizer of functions. *SIAM Journal on Scientific and Statistical Computing*, 6:15–29, 1985.

[6] A. H. G. Rinnooy Kan and G. T. Timmer. A stochastic approach to global optimization. In P. Boggs, R. Byrd, and R. B. Schnabel, editors, *Numerical Optimization*, pages 245 – 262, SIAM, Philadelphia, 1984.