CONCURRENT FUNCTION EVALUATIONS IN LOCAL AND GLOBAL OPTIMIZATION

Robert B. Schnabel

CU-CS-345-86 October 1986

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Research supported by Air Force grant AFOSR-85-0251, ARO contract DAAG 29-84-K-0140,and NSF cooperative agreement DCR-84220944.

Abstract

This paper discusses some basic opportunities for the use of multiprocessing in the solution of optimization problems. We consider two fundamental optimization problems, unconstrained optimization and global optimization, in the important case when function evaluation is expensive and gradients are evaluated by finite differences. First we discuss some simple parallel strategies based upon the use of concurrent function evaluations to evaluate the finite difference gradient. These include the speculative evaluation of the gradient concurrently with the evaluation of the function, before it is known whether the gradient value at this point will be required. We present examples that indicate the effectiveness of these parallel strategies for unconstrained optimization. We also give experimental results that show the effect of using these strategies to parallelize each of the multiple local minimizations within a recently proposed concurrent global optimization algorithm. We briefly discuss several parallel optimization strategies that are related to these approaches but make more fundamental changes to standard sequential optimization algorithms.

1. Introduction

The recent proliferation of multiprocessor computers presents many opportunities and challenges for numerical computation. This paper discusses some basic opportunities for the use of multiprocessing in one area of numerical computation, the solution of optimization problems. We consider two basic optimization problems, unconstrained optimization and global optimization, in the important case when function evaluation is expensive. We discuss some simple but effective parallel methods, and briefly propose several more complex strategies that appear to merit investigation.

The most basic nonlinear optimization problem is the *unconstrained optimization* problem,

$$\min_{x \in \mathbf{R}^n} f : \mathbf{R}^n \rightarrow \mathbf{R} \tag{1.1}$$

where $f(x)$ is at least twice continuously differentiable. Usually it is sufficient to find a *local minimizer* of $f(x)$, a point where $f(x)$ attains its lowest value over some open neighborhood of the variable space. In this paper we will refer to this problem as the *unconstrained local optimization* problem. Sometimes it is necessary to find the lowest among multiple local minimizers of $f(x)$. This problem is referred to as the *global optimization* problem. When discussing global optimization in this paper, we will make the common assumption that a rectangular region S defined by lower and upper bounds on each variable is given, i.e.

$$S = \{x \mid l_i \leq x_i \leq u_i \ , \ i = 1, \ . \ . \ . \ , n\} \tag{1.2}$$

and that the global minimizer and all local minimizers (of interest) are known to lie in the interior of S.

Both local and global optimization problems often are expensive to solve. The main reason is that the objective function, $f(x)$, often is itself a complex computer code, for example a flight or circuit simulation. It is not unusual for each evaluation of $f(x)$ to require many seconds, or minutes, on a powerful computer. Since the solution to the optimization problem usually requires many evaluations of $f(x)$, it becomes an expensive process.

In many instances when $f(x)$ is expensive, the derivatives of $f(x)$ are not available analytically. We assume throughout this paper that this is the case. In this case, optimization codes approximate the gradient of $f(x)$ at a point x_c by using the finite difference approximation

$$\nabla f(x_c)_i = \frac{f(x_c + h_i e_i) - f(x_c)}{h_i} \tag{1.3}$$

where h_i is a small stepsize and e_i denotes the i^{th} unit vector. Thus each gradient evaluation requires n function evaluations in addition to $f(x_c)$. No higher derivatives usually are used when $f(x)$ is expensive. Therefore, the cost of solving optimization problems with expensive function evaluations and no analytic gradients often is dominated by the function and finite difference gradient evaluations. If the number of variables is not too large, say $n \leq 50$, then the time required by the remainder of the optimization algorithm often is insignificant in comparison.

Due to the expensive nature of many optimization problems, there is ample incentive to devise methods for solving them on parallel computers if they can lead to significantly faster or more cost effective solution of these problems. For problems with expensive function evaluations as discussed above, there are two obvious types of approaches. One can use standard sequential optimization methods but apply a parallel algorithm to evaluate $f(x)$, or one can devise methods that make effective use of evaluating $f(x)$ at multiple points concurrently. In this paper we discuss the later approach. The former approach, applying a parallel algorithm to evaluate $f(x)$, is dependent upon the actual objective function $f(x)$ and is not under the control of the optimization algorithm designer. It should be noted, however, that the two approaches often are quite compatible. For example, in many cases the evaluation of $f(x)$ vectorizes well. In this case, a computer such as the Cray X-MP or Alliant, which consist of multiple vector processors, would allow multiple evaluations of $f(x)$ to be performed concurrently with each evaluation performed by a vector processor. In other cases where the computation of $f(x)$ requires multiple independent processors, it would be possible to divide the processors of a multiprocessor into groups, with each group of processors used to evaluate $f(x)$ and the multiple groups used to implement the parallelism discussed in this paper.

The approach we discuss in this paper, the concurrent evaluation of $f(x)$ at multiple points, is well suited to any computer that can execute multiple, different instruction streams concurrently. Such machines are known as *Multiple Instruction Multiple Data* (MIMD) computers. This class includes both shared memory multiprocessors, and local memory multiprocessors such as hypercubes; the algorithms we discuss are well suited to any such computer. Our approach is not generally suited to *Single Instruction Multiple Data* (SIMD) computers, such as processor arrays, whose processors can execute the same instruction on different data in lock-step. This is because different evaluations of an expensive function $f(x)$ generally require different sequences of instructions, due to data dependent branches in the code for $f(x)$, and thus cannot easily be performed concurrently on an SIMD machine.

In Section 2 we discuss the use of concurrent function evaluations in unconstrained local optimization. The most obvious idea is to perform the n function evaluations required by the finite difference gradient (1.3) in parallel. We discuss several variations on this idea, and show how their effectiveness depends upon the number of processors available and an important problem characteristic, the ratio of function to gradient evaluations. We then briefly discuss several more sophisticated strategies.

Section 3 discusses the use of concurrent function evaluations in global optimization. Byrd, Dert, Rinnooy Kan, and Schnabel [1986] have proposed and tested a concurrent global optimization algorithm in which local minimizations are performed sequentially. First we very briefly summarize this algorithm and its performance. Then we discuss the incorporation of the concurrent local optimization techniques of Section 2 into this global optimization algorithm, and present some experimental results.

2. Concurrent Function Evaluation in Unconstrained Local Optimization

A variety of algorithms exist for finding a local minimizer of a continuously differentiable function without constraints. (See e.g. Fletcher [1980], Gill, Murrary and Wright [1981], or Dennis and Schnabel [1983].) When function evaluation is expensive, analytic derivatives are unavailable, and n is not too large (say less than or equal to 100), the most commonly used method is a quasi-Newton method utilizing finite difference gradients, BFGS updates to approximate the Hessian, and a line search. Such an algorithm is outlined in Algorithm 2.1.

When function evaluation is expensive and n is not too large, the cost of Algorithm 2.1 is dominated by the function and finite difference gradient evaluations shown in boldface. The most important point to be noted from Algorithm 2.1 is the order in which these occur: one or more function evaluations always precede exactly one gradient evaluation. In our practical experience, there rarely is more than one gradient evaluation per iteration, and the average number of function evaluations per gradient evaluation, or per iteration, is often quite close to one, say around 1.25. On very hard problems the number of function evaluations per iteration might be as high as 2. A typical pattern of function and gradient evaluations might be

Algorithm 2.1. Quasi-Newton Method for Unconstrained Optimization

```
Given initial iterate $x_c$ , $f_c = f(x_c)$ , $g_c =$ finite difference approximation to $\nabla f(x_c)$ ,
    $H_c \in \mathbf{R}^{n \times n}$ positive definite (e.g. $H_c = I$)

{ calculate search direction }
    solve $H_c d = -g_c$ for $d$        { $d$ is search direction }

{ line search }
    $\lambda := 1$                        { $\lambda$ is steplength }
    linesearchdone := false,  $\lambda$min := 0,   $\lambda$max := $\infty$

    repeat
        calculate $f_+ = f(x_c + \lambda d)$
        if $f_+$ is sufficiently $< f_c$ then
            calculate $g_+ =$ finite difference approximation to $\nabla f(x_c + \lambda d)$
            if $g_+^T d$ is sufficiently $> g_c^T d$
                then linesearchdone := true
                else $\lambda$min := $\lambda$ , increase $\lambda$     { new $\lambda \in$ ( old $\lambda$ , $\lambda$max) }
        else     { $f_+$ too large }
            $\lambda$max := $\lambda$ , reduce $\lambda$    { new $\lambda \in$ ( $\lambda$min, old $\lambda$ ) }
    until linesearchdone

{ update function and derivative information }
    decide whether to stop ; if not :
        $x_c := x_c + \lambda d$ , $f_c := f_+$, $g_c := g_+$

        $H_c := H_c - \frac{H_c d d^T H_c}{d^T H_c d} + \frac{(g_+ - g_c)(g_+ - g_c)^T}{\lambda_c d^T (g_+ - g_c)}$        { BFGS update }
        begin next iteration    (go to first line of algorithm)
```

$$f, f, g, f, g, f, g, f, f, f, g, f, g, f, g, f, g, f, g, f, g \ . \tag{2.1}$$

The most obvious way to parallelize Algorithm 2.1 when function evaluation is expensive and the gradient is calculated by finite differences (as we are assuming in this paper) is to perform the n function evaluations for the finite difference gradient concurrently. If the number of processors, p, is greater than or equal to n, this requires one *concurrent function evaluation* *step,* a step where all (or some) processors perform a function evaluation concurrently. If $p<n$, this will require $\lceil n/p \rceil$ concurrent function evaluation steps. For example, if $n = 6$ and $p = 4$, the beginning of the sequence given in (2.1) would be performed as shown in Example 2.2a below. Here g_i denotes the i^{th} function evaluation for the finite difference gradient calculation, and a dash indicates that the processor is idle. The letter f denotes an evaluation of $f(x)$ at a trial point $x_c+\lambda d$; we will refer to this as a *trial point function evaluation,* to differentiate it from the n function evaluations performed for each finite difference gradient.

For ease of reference, the first three simple parallel modifications of Algorithm 2.1 that we discuss in this section are summarized in Table 2.3. The strategy we have just discussed is

Example 2.2a -- Parallel Strategy I Applied to Sequence f,f,g,f,g,f,g

	Concurrent Step Number									
	1	2	3	4	5	6	7	8	9	10
Processor 1	f	f	g_1	g_5	f	g_1	g_5	f	g_1	g_5
Processor 2	-	-	g_2	g_6	-	g_2	g_6	-	g_2	g_6
Processor 3	-	-	g_3	-	-	g_3	-	-	g_3	-
Processor 4	-	-	g_4	-	-	g_4	-	-	g_4	-

Example 2.2b -- Parallel Strategy II Applied to Sequence f,f,g,f,g,f,g

	Concurrent Step Number						
	1	2	3	4	5	6	7
Processor 1	f	f	g_4	f	g_4	f	g_4
Processor 2	g_1	g_1	g_5	g_1	g_5	g_1	g_5
Processor 3	g_2	g_2	g_6	g_2	g_6	g_2	g_6
Processor 4	g_3	g_3	-	g_3	-	g_3	-

referred to as Parallel Strategy I. Note that Parallel Strategy I does not use multiple processors during the trial point function evaluations.

If we assume that all the expense in Algorithm 2.1 is in the function and finite difference gradient calculations, and that each function evaluation takes the same amount of time, then the effectiveness of Parallel Strategy I (and Parallel Strategies II and III) depends only on three

Table 2.3. Simple Parallel Modifications of Algorithm 2.1

n = number of variables, p = number of processors

Parallel Strategy I -- Parallel Finite Difference Gradients

Each time $\nabla f(x)$ is required, perform the n function evaluations for the finite difference gradient concurrently ($\lceil n/p \rceil$ concurrent function evaluation steps)

Each trial point function evaluation $f(x_c+\lambda d)$ is performed by one processor with the remaining processors idle

Parallel Strategy II -- Parallel Finite Difference Gradients with Partial Speculative Gradient Evaluation

Each time a trial point function value $f(x_c+\lambda d)$ is required, perform $\min\{p-1, n\}$ function evaluations for the finite difference gradient at $x_c+\lambda d$ concurrently with the evaluation of $f(x_c+\lambda d)$ (one concurrent function evaluation step)

If $p<n+1$, then each time $\nabla f(x_c+\lambda d)$ is required, calculate the $n+1-p$ remaining function evaluations for the finite difference gradient concurrently ($\lceil (n+1-p)/p \rceil$ concurrent function evaluation steps)

Parallel Strategy III -- Parallel Finite Difference Gradients with Full Speculative Gradient Evaluation

Each time a trial point function value $f(x_c+\lambda d)$ is required, perform the n function evaluations for the finite difference gradient at $x_c+\lambda d$ concurrently with the evaluation of $f(x_c+\lambda d)$ ($\lceil (n+1)/p \rceil$ concurrent function evaluation steps) problem and algorithm characteristics. These are the number of variables, n, the number of processors, p, and the ratio of the number of trial point function evaluations to the number of finite difference gradient calculations. If Algorithm 2.1 uses f trial function evaluations and g gradient evaluations, then a sequential implementation requires $f+ng$ function evaluations, while the parallel finite difference gradient approach requires $f+(\lceil n/p \rceil)g$ concurrent function evaluations steps. Thus the speedup, defined in general by

$$\text{speedup} = \frac{\text{time required by sequential algorithm}}{\text{time required by parallel algorithm}} ,$$

is given for Parallel Strategy I by

$$\text{speedup}_I = \frac{(f/g) + n}{(f/g) + \lceil n/p \rceil} . \tag{2.2}$$

Table 2.4a gives an indication of the effectiveness of the parallel finite difference gradient strategy. For a moderate problem size, $n = 25$, it shows the efficiency for five values of f/g, 1, 1.25, 1.5, 2, 3, and for various numbers of processors. Recall that the values 1.25 or perhaps 1.5 are typical f/g ratios, 2 is unusual and 3 extremely rare. Table 2.4 uses powers of two for the number of processors because on some computers, such as hypercubes, these are the only possibilities. In parentheses we indicate the smallest number of processors which would give the same speedup; this number is applicable to computers when any subset of the processors may be used.

Several important, general points are illustrated by Table 2.4a. First, when p is considerably less than n, even the very simple parallel finite difference gradient strategy achieves almost optimal performance. This is because each gradient evaluation requires $\lceil n/p \rceil$ concurrent function evaluation steps, which parallelize almost perfectly, and this cost dominates the f trial point evaluation steps which each use only one processor.

Table 2.4a. Speedup of Parallel Strategy I in Comparison to Sequential Algorithm 2.1 when $n = 25$

	Function Evaluations (not counting finite difference gradients)/ Gradient Evaluations				
	1	1.25	1.5	2	3
# Processors					
4 (4)	3.25	3.18	3.12	3.00	2.80
8 (7)	5.20	5.00	4.82	4.50	4.00
16 (13)	8.67	8.08	7.57	6.75	5.60
32 (25)	13.00	11.67	10.60	9.00	7.00

Table 2.4b. Speedup of Parallel Strategy II in Comparison to Sequential Algorithm 2.1 when $n = 25$

	Function Evaluations (not counting finite difference gradients)/ Gradient Evaluations				
	1	1.25	1.5	2	3
# Processors					
4 (4)	3.71	3.62	3.53	3.38	3.11
8 (7)	6.50	6.18	5.89	5.40	4.67
16 (13)	13.00	11.67	10.60	9.00	7.00
32 (26)	26.00	21.00	17.67	13.50	9.33

Table 2.4c. Speedup of Parallel Strategy III in Comparison to Sequential Algorithm 2.1 when $n = 25$

	Function Evaluations (not counting finite difference gradients)/ Gradient Evaluations				
	1	1.25	1.5	2	3
# Processors					
4 (4)	3.71	3.00	2.52	1.93	1.33
8 (7)	6.50	5.25	4.42	3.38	2.33
16 (13)	13.00	10.50	8.83	6.75	4.67
32 (26)	26.00	21.00	17.67	13.50	9.33

Secondly, when $p = n$, the maximum speedup that can be attained using the simple parallel finite difference gradient strategy is $(p+1)/2$, about half of the optimal. This speedup will occur if $f=g$; if $f \geq g$, the speedup is lower. The problem is that the trial point function evaluations are performed sequentially, with $n-1$ processors idle, and only the finite difference gradient evaluations are performed concurrently requiring one concurrent function evaluation step per gradient value. Thus $n-1$ processors are idle at least half of the time. The strategies discussed in the remainder of this section address this problem.

The parallel finite difference gradient strategy never utilizes more than n processors. If $p \geq n$, $p-n$ processors are unused. This deficiency is shared by the next few parallel strategies we discuss which use a maximum of $n+1$ processors; at the end of this section we discuss ways to use more than $n+1$ processors.

Now we consider ways to make use of multiple function evaluations during the evaluation of $f(x_c+\lambda d)$ in Algorithm 2.1. One way is to evaluate $f(x)$ at multiple points on the line in the direction d from x_c, and perhaps in additional search directions, concurrently with the evaluation of $f(x_c+\lambda d)$, and take the best of these points as the next iterate. This possibility has been considered by Dixon [1981], Patel [1982], and Lootsma [1984]. Note that this strategy alters the optimization algorithm, as opposed to Parallel Strategy I which just reorders the calculations of Algorithm 2.1. As one might expect, given the fact that the first trial point in the line search usually is acceptable, the speedup from utilizing $p-1$ additional points in the line search is not nearly proportional to p, in fact usually it is quite small. Therefore we are interested in considering other strategies.

A simpler way to utilize additional processors during the trial point function evaluations is to always perform $p-1$ function evaluations that would be required for the finite difference value of $\nabla f(x_c+\lambda d)$ concurrently with the evaluation of $f(x_c+\lambda d)$. We refer to this as a *speculative partial gradient evaluation.* If $f(x_c+\lambda d)$ is acceptable so that the gradient at $x_c+\lambda d$ is required, as is the case most of the time, then only $n+1-p$ function evaluations remain for the finite difference gradient calculation (none if $p \geq n+1$). This means that one concurrent function evaluation step is saved per gradient evaluation in comparison to Parallel Strategy I, unless n is a multiple of p in which case the costs of the two algorithms are the same. If $f(x_c+\lambda d)$ is too high and the gradient at $x_c+\lambda d$ is not required, the $p-1$ speculative components of $\nabla f(x_c+\lambda d)$ that have been computed are wasted in the context of Algorithm 2.1, but the cost is the same as if the $p-1$ processors had been left idle.

We refer to the strategy of using the remaining $p-1$ processors to make a speculative partial gradient evaluation in conjunction with each trial point function evaluation as Parallel Strategy II. It is always at least as efficient as the simple parallel finite difference strategy, Parallel Strategy I. As an illustration, Example 2.2b shows the performance of Parallel Strategy II on the same example as was used for Parallel Strategy I in Example 2.2a. Parallel Strategy II requires three fewer concurrent function evaluation steps than Parallel Strategy I in this case. The only unnecessary speculative function evaluations are the evaluations of g_1, g_2, and g_3 at the first step. Towards the end of this section we discuss ways to use these "wasted" speculative gradient values.

The cost of Parallel Strategy II, using the notation introduced above, is $f + (\lceil (n+1)/p \rceil - 1)\, g$ concurrent function evaluation steps. Thus the speedup of Parallel Strategy II in comparison to the sequential Algorithm 2.1 is

$$\text{speedup}_{II} = \frac{(f/g) + n}{(f/g) + \lceil (n+1)/p \rceil - 1} \quad . \tag{2.3}$$

Table 2.4b illustrates the speedup of Parallel Strategy II in the same cases used in Table 2.4a.

Table 2.4b shows that the simple strategy of using parallel finite difference gradients including speculative partial gradient evaluation is very effective when $p \leq n$. When $p = (n+1)/2$, the efficiency is 80-90% of optimal for the usual f/g ratios, and for $p = n+1$ it is still 70-80% of optimal. Most users of parallel computers would be quite satisfied with such utilization, and it is not clear how much more parallelism can be gained when $p \leq n+1$, although we suggest some possibilities shortly. The most important limitation of Parallel Strategy II is that it can not utilize more than n + 1 processors.

Another parallel variant of Algorithm 2.1 would be to evaluate the *entire* gradient whenever a trial point function evaluation is made. That is, whenever $f(x_c+\lambda d)$ is evaluated, $p-1$ function evaluations for the finite difference $\nabla f(x_c+\lambda d)$ are performed concurrently and then, if $p \leq n$, the remaining $n+1-p$ function evaluations for $\nabla f(x_c+\lambda d)$ are performed using $\lceil (n+1)/p \rceil -1$ additional concurrent function evaluation steps. We refer to this strategy as *speculative full gradient evaluation,* and the strategy that uses it is called Parallel Strategy III. It should be clear that Parallel Strategies II and III are identical when $p \geq n+1$, but that Parallel Strategy III is less efficient than Parallel Strategy II whenever $p < n+1$ because it takes extra steps to calculate the remainder of the finite difference gradient whether it will be needed or not. There are several reasons why we still mention this strategy. They include that it is more practical than Parallel Strategy II in the context of global optimization, and that it may be worthy of consideration in conjunction with the more complex strategies discussed shortly.

Because Parallel Strategy III makes a complete gradient evaluation at every trial point, its cost is simply $(\lceil (n+1)/p \rceil)f$ concurrent function evaluation steps. Its speedup in comparison to the sequential Algorithm 2.1 is

$$\text{speedup}_{III} = \frac{(f/g) + n}{\lceil (n+1)/p \rceil} \quad . \tag{2.4}$$

Table 2.4c illustrates the speedup of this approach. Note that the speedups in Table 2.4c are the same as those for Parallel Strategy II in Table 2.4b when $p \geq n+1$ or $f/g = 1$, and are smaller otherwise. In fact, comparison with Table 2.4a shows that speculative full gradient evaluation is inferior to the simple parallel finite difference gradient strategy for small enough p or large enough f/g; comparison of Equations 2.2 and 2.4 shows that Parallel Strategy I is superior if and only if

$$f/g + \lceil n/p \rceil \leq (f/g)\,(\lceil (n+1)/p \rceil) \quad . \tag{2.5}$$

For example, if $f/g = 1.25$, and $n \neq 0 \bmod p$, Parallel Strategy I is superior if $p < n/5$.

From the point of view of the optimization algorithm, the three parallel strategies discussed so far in this section do not make any real change to Algorithm 2.1. Parallel Strategy I is identical to Algorithm 2.1 except that the finite difference gradient is computed concurrently. In Parallel Strategies II and III, the speculative gradient evaluations result in the calculation of some gradient values that are not calculated by the sequential algorithm (at the unsuccessful trial points) but no use is made of this information. For any particular problem, all three parallel algorithms will produce the same sequence of iterates as the sequential algorithm.

It would be more interesting, and possibly more beneficial, to consider strategies that actually solve the optimization problem differently due to the use of multiple processors and concurrent function evaluations. In the remainder of this section we indicate several such possibilities, of two general types. The first group of parallel algorithms we discuss are applicable when there are $n+1$ (or fewer) processors, the second when they are more than $n+1$ processors.

When the number of processors is less than or equal to $n+1$, Parallel Strategy II makes almost optimal use of multiple processors as along as f/g is not much greater than 1. Since this is generally the case, it seems reasonable to apply this basic strategy and see whether any additional advantage can be gained through it. The main opportunity is to try to make use of the (partial) gradient values at the unsuccessful trial points, which are available in the parallel algorithm but not in the sequential version. We now consider several uses that can be made of this information. For simplicity we consider only the case $p = n+1$, meaning that the full gradient is available at each trial point.

The simplest way to utilize the gradient value at an unsuccessful trial point would be to continue the line search in the direction d as usual, and use the gradient value to help calculate the next value for the step length parameter λ. Indeed, line search algorithms that utilize gradient values from all trial points are well known (see Gill, Murray, and Wright [1981] p. 104 for several references). In our experience they do not greatly improve the optimization algorithm's efficiency. In any case, only the directional derivative, $\nabla f(x_c+\lambda d)^T d$, is needed and this can be calculated using just one extra function evaluation as opposed to the n function evaluations required for the full gradient. So this strategy requires only two processors, not $n+1$.

A second approach would be to again continue the line search in the direction d after the function and gradient at the unsuccessful trial point have been evaluated. After the line search is completed, the algorithm would have least three function and three gradient values along the line from the old to the new iterate. This is enough information to form a more complex model than the standard quadratic model around the new iterate. The following iterate would then be based upon minimizing this model. The hope is that this would lead to a better next iterate than would be obtained from the standard quadratic model, and that the total number of iterations required to solve the optimization problem would be reduced. One type of model that could be used is a tensor model, introduced by Schnabel and Frank [1984]. A line search with even one unsuccessful trial point and the gradient value at each point enables the formation of a fourth order tensor model that interpolates three gradient and two function values, and the minimum of a such a model can be calculated efficiently.

A third approach would be to use the gradient value at the unsuccessful trial point $x_c+\lambda d$ to immediately alter the Hessian approximation H_c at x_c and the search direction d from x_c. In a quasi-Newton algorithm, it is possible that the objective function $f(x)$ is nearly quadratic, but that the trial point $x_c+\lambda d$ still is bad because H_c is inaccurate. Indeed, if $f(x)$ is a positive definite quadratic, the function and gradient values at x_c and any $x_c+\lambda d$ are sufficient to determine the minimum x_+ of $f(x)$ along the line in the direction d from x_c, and the new BFGS Hessian approximation H_+ at x_+, *without* calculating $f(x_+)$ or $\nabla f(x_+)$. This is because any three pieces of function and gradient information along a line determine the minimizer of a quadratic along that line, and any two gradient values along a line determine the unique secant equation of the quadratic along that line, and hence the BFGS update which is the same at all points on that line. Furthermore, since the secant equation $H_+(x_+-x_c) = g_+ - g_c$ implies that $(x_+ - H_+^{-1}g_+) = (x_c - H_+^{-1}g_c)$, it is not necessary to know the next iterate x_+ to calculate the trial value of the succeeding iterate; it is equivalent to update H_c to H_+ using the gradient information from the unsuccessful trial point, calculate $H_+^{-1}g_c$, and continue iterating from x_c.

For quadratic $f(x)$, this strategy would allow a BFGS algorithm where the gradient value is available at every trial point to always use only one trial point per iteration. Thus it may save function evaluations in comparison to Algorithm 2.1. We are now investigating ways to adapt such techniques to general $f(x)$. It is clear that one would not always want to use the above strategy; one would need to determine when $f(x)$ is sufficiently close to quadratic that this approach is likely to be beneficial. One possibility would be to tentatively update H_c to H_+ at x_c using the secant equation from $\nabla f(x_c+\lambda d)$ and $\nabla f(x_c)$, and then see whether the new quadratic model predicts $f(x_c+\lambda d)$ reasonably well. If so, the new line search direction $H_+^{-1}\nabla f(x_c)$ would be used from x_c, otherwise the update would be rescinded and the line search continued in the old direction d. A more complex strategy would be needed if multiple iterations of the type described above were allowed.

The three approaches we have just discussed can utilize at most $n+1$ processors. There are many expensive optimization problems where the number of variables is not very large, say $n \leq 25$. Thus it is likely that for many expensive optimization problems, the number of processors available on many multiprocessors, p, will exceed $n+1$.

If $p \geq (n^2 + 3n + 2)/2$, then it is possible to evaluate the function, the finite difference gradient, and a finite different Hessian approximation simultaneously. A very likely situation, however, is that $p \in ((n+1), (n^2 + 3n + 2)/2)$, so that there are more than enough processors to evaluate the finite difference gradient but not enough to calculate the full finite difference Hessian as well. For example, if $p = 64$, any problem with $n \in [10, 63]$ falls in this class. In the remainder of this section we briefly discuss some approaches for this case.

An obvious extension of the techniques discussed above to the case where there are more than $n+1$ processors would be to evaluate the function at the trial point $x_c+\lambda d$, the finite difference gradient at $x_c+\lambda d$, and some part of the finite difference Hessian simultaneously. Two important issues are whether the finite difference Hessian information should be evaluated at $x_c+\lambda d$ or at x_c , and how to form the new Hessian approximation using this partial finite difference Hessian information.

Calculating a partial finite difference Hessian approximation at $x_c+\lambda d$ while evaluating the function and gradient at $x_c+\lambda d$ would provide the most useful information if $x_c+\lambda d$ is accepted as the next iterate. If $x_c+\lambda d$ is not accepted, however, it might not be possible to make good use of this Hessian information. An alternative would be to calculate some partial finite difference Hessian information at x_c while calculating the function and gradient at $x_c+\lambda d$. If $x_c+\lambda d$ is accepted, then the Hessian information could be used to improve the Hessian approximation at x_c before applying the BFGS update to obtain the new Hessian approximation at $x_c+\lambda d$. If $x_c+\lambda d$ is not accepted, the Hessian approximation at x_c still could be updated and the line search direction d revised in a manner similar to the final $n+1$ processor strategy discussed above.

Perhaps the most challenging issue in developing an unconstrained optimization algorithm that makes use of more than $n+1$, but fewer than $(n^2 + 3n + 2)/2$, function evaluations at each iteration is how best to use these function evaluations to form the Hessian approximation at a point x. One possibility would be to use the $p-(n+1)$ processors that are available (after allocating $n+1$ for the function and finite difference gradient) to calculate $k = \lfloor (p-(n+1))/n \rfloor$ finite difference gradient values at points that are close to x and in mutually orthogonal (or conjugate) directions s_i from it. This information would give k secant equations $H_c s_i = y_i$, $i = 1, \ldots, k$. The Hessian approximation could then be updated using the multiple secant update techniques of Schnabel [1983]; probably only the equations consistent with positive definiteness should be used. If the directions s_i are properly chosen, then it appears that such an approach would lead to a $\lceil n/k \rceil$ step locally q-quadratically convergent method. By incorporating the standard secant equation as well, one step q-superlinear convergence might also be retained. An alternative approach would be to use the additional function evaluations to calculate some components of the Hessian directly using second order finite differences. This would be more efficient if $(n^2+3n+2)/2$ processors are available, but if fewer processors were available it might be difficult to update the Hessian element by element while still retaining positive definiteness.

It should be clear from the above discussion that there are many interesting challenges in utilizing multiple processors for unconstrained local optimization. While all the techniques mentioned above are inspired by the consideration of parallel processing, it is possible that some of them could lead to improved sequential algorithms as well.

3. Concurrent Function Evaluation in Global Optimization

Several types of sequential methods.have been proposed for finding the global minimum of a function $f(x)$ within the rectangular region S defined by (1.2). These include deterministic methods, stochastic methods, and methods that utilize interval arithmetic. It is beyond the scope of this paper to review these methods; papers describing several leading modern methods, including the tunneling method (Levy and Gomez [1985]), several stochastic methods (Rinnooy Kan and Timmer [1985]), and an interval arithmetic method (Walster, Hansen, and Sengupta [1985]) can be found in Boggs, Byrd, and Schnabel [1985]. We are not aware of any computational studies that have established whether any of these methods are superior to any of the others in general.

One of the promising approaches to global optimization appears to be the multi level single linkage algorithm of Rinnooy Kan and Timmer [1984]. Byrd, Dert, Rinnooy Kan, and Schnabel [1986] have proposed a concurrent variant of this method. The algorithm is outlined in Algorithm 3.1 below.

Algorithm 3.1 -- A Concurrent Multi-Level Single Linkage Method for Global Optimization

Given $f : \mathbf{R}^n \rightarrow \mathbf{R}$, feasible region S and p processors

```
0. Partition S
      Subdivide $S$ into $p$ equal size, regular shaped subregions $S_i$, $i = 1,...p$, and assign subregion $S_i$ to processor $i$ for $i = 1,...,p$.

At iteration number $k$:

1. Generate sample points and function values
      For $i = 1,...,p$
            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
      For $i = 1,...,p$
            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 (some start points selected above may be eliminated).

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.
```

Algorithm 3.1 is intended to make efficient use of multiple processors whether or not the evaluation of the objective function $f(x)$ is expensive. If $f(x)$ is expensive, as we assume throughout this paper, then only the costs of Steps 1 and 3 are important since all the function and derivative evaluations occur in these steps. The cost of Step 2 is important for problems where $f(x)$ is inexpensive but becomes inconsequential as function evaluation becomes expensive.

It is typical to evaluate $f(x)$ at between 100 and 1,000 points in Step 1 at each iteration. Thus, as long as each function evaluation takes the same amount of time, Step 1 can make optimal use of at least 100 processors.

It is not as easy to make efficient use a large number of processors in Step 3. For many global optimization problems, the number of local minimizations at each iteration can be expected to be small, sometimes 10 or fewer. If each of these local minimizations is solved by a sequential algorithm, then the number of processors that can be utilized is at most equal to the number of minimizations. Furthermore, the local minimizations tend to be of differing lengths, so that not even this number of processors can be fully utilized. For example, in one of our runs of the Hartman 3 test problem, for which n=3, using 1000 sample points per iteration, 4 local minimizations were performed at the first iteration with the following pattern of function and gradient evaluations:

$$\begin{array}{l}
f\ g\ f\ f\ f\ f\ g\ f\ g\ f\ g\ f\ g\ f\ g \\
f\ g\ f\ f\ g\ f\ f\ g\ f\ f\ f\ f\ g\ f\ g\ f\ g\ f\ g\ f\ g\ f\ g \\
f\ g\ f\ f\ f\ f\ g\ f\ g\ f\ g\ f\ g\ f\ g\ f\ g\ f\ g\ f\ g\ f\ g\ f\ g \\
f\ g\ f\ f\ f\ g\ f\ g\ f\ g\ f\ g\ f\ g
\end{array} \qquad (3.1)$$

These minimizations require a total of 46 function and 33 gradient evaluations, or 145 function evaluations if the gradients are calculated by finite differences. If the 4 minimizations are performed concurrently, each using one of 4 processors, then since the longest search requires 47 function evaluations, a speedup of 3.1 is obtained. It is not possible to use more processors as long as each minimization is performed sequentially. We return to this example later in this section.

Byrd, Dert, Rinnooy Kan, and Schnabel [1986] have tested Algorithm 3.1 on a standard set of seven global optimization problems. These problems all are small, with n between 2 and 6, and the number of local minimizers between 3 and 10. From their results they can simulate the speedup that would be obtained on these problems using any number of processors, when only the cost of function evaluations is being considered. Of course, the speedup depends upon the sample size in Step 1. A sample size of 1000 points per iteration may be indicative of practical sample sizes for harder problems. With this sample size, the speedup with 8 processors was between 6.0 and 7.1 for the seven test problems; with 16 processors, between 8.9 and 11.2; with 32 processors, between 12.0 and 15.9; and with 1000 processors, between 17.9 and 27.0. If a sample size of 200 is used, the local minimization step consumes a proportionally greater amount of time and the speedups are lower because the minimization step doesn't parallelize as well as the sampling step. The speedups are between 3.5 and 6.1 with 8 processors, between 4.1 and 8.6 with 16 processors, between 4.4 and 9.6 with 32 processors, and between 4.8 and 10.0 with 200 (or 1000) processors.

These results illustrate that is rather easy to make good use of a relatively small number of processors for global optimization problems where function evaluation is expensive, but that the speedup for larger numbers of processors may be limited due to the small number of local minimizations in Step 3. In fact if Step 2, the start point selection step, performs ideally, then the number of local minimizations will equal the number of local minimizers. Thus it may not be possible to utilize more than this number of processors in Step 3 if the local minimizations are performed sequentially.

From the above discussion, it is clear that the parallel local minimization strategies discussed in Section 2 are of interest within the context of global optimization as long as the gradient evaluations are performed by finite differences. This usually is the case on problems with expensive function evaluations. In this case, the strategies of Section 2 present the opportunity of using $n+1$ times as many processors in the local minimization step of the global optimization algorithm, and thus obtaining up to $n+1$ times greater speedup in this step.

We have investigated the effects of applying the simple concurrent strategies discussed in the beginning of Section 2 to the local minimizations in Step 3 of our concurrent global optimization algorithm. The best strategy, Parallel Strategy II, is difficult to apply in the context of global optimization because it is not evident how many processors to allocate for a particular partial gradient evaluation when other local minimizations are being performed concurrently. Therefore we have tested Parallel Strategy I, which performs the trial point function evaluations sequentially and the finite difference gradients in parallel, and Parallel Strategy III, which performs a full speculative finite difference gradient evaluation along with each trial point function evaluation.

We calculated the speedups in the local minimization phase (step 3) that result from applying Parallel Strategies I and III to step 3 of the first iteration of Algorithm 3.1. This was done for each of the seven test problems, each run with four different sample sizes. (Different sample sizes lead to different numbers and lengths of local minimizations.) Thus there are 28 test cases, each a set of local minimizations like (3.1). For each test case, we calculated the speedups with the number of processors, p, equal to $nm/4$, $nm/2$, nm, and $2nm$, where m is the number of local minimizations at the first iteration of Algorithm 3.1. When $p = 2nm$, there are more than enough processors to conduct a parallel finite difference gradient evaluation for each search simultaneously, if required. In the other cases, there are not, so a queue of function evaluation requests is maintained, and if the number of function evaluation requests exceeds the number of processors at any particular time, then the function evaluations are processed in the order they are received.

Table 3.2 summarizes the average utilization of processors in *only the local minimization phase* of iteration 1 of Algorithm 3.1, for the sequential and two parallel local minimization strategies. For each local minimization strategy and each number of processors, we give the average *efficiency* in the local minimization phase over the 28 test sets, where the efficiency is defined by

$$\text{efficiency} = \frac{\text{time required by sequential algorithm}}{p \times (\text{ time required by parallel algorithm })} = \frac{\text{speedup}}{p} .$$

It is necessary to use this measure rather than speedup because the number of processors represented by a particular line of Table 3.2 may differ from problem to problem since the number of local minimizations may be different for each test case.

It makes less sense to calculate the average gain in the speedup of the entire concurrent global optimization algorithm from parallelizing the individual local minimizations, because this depends crucially upon the relative number of function evaluations in the sampling and minimization steps. Instead, we consider as an example the run of the Hartman 3 test problem

Table 3.2. Average Efficiency of Local Minimization Step of Concurrent Global Optimization Algorithm

n = number of variables, m = number of local minimizations

	Each Local Minimization Performed by:		
	Sequential	Parallel Strategy I	Parallel Strategy III
Number of Processors			
nm/4	.76	.93	.76
nm/2	.48	.76	.73
nm	.26	.48	.63
2nm	.13	.24	.41

with a 1000 sample points, whose minimization step is shown in (3.1). The overall speedups for this problem with 6, 12, 24, and 1000 processors, and with the individual local minimizations performed sequentially or by Parallel Strategies I or III, are given in Table 3.3. We note that if only 200 sample points are used, then there are more local searches, 7, and the effects are more dramatic. For example for $p=20$ and 200 sample points, the speedups with local minimizations performed sequentially, by Parallel Strategy I, and by Parallel Strategy III are 7.4, 11.3, and 14.1, respectively.

The results in Tables 3.2 and 3.3 illustrate that considerable additional speedups can be obtained by applying the simple parallel strategies of Section 2 to each local minimization subproblem in a concurrent global optimization algorithm that conducts multiple local minimizations simultaneously. Table 3.2 shows that Parallel Strategy I achieves very good utilization of the processors when the number of processors are small in comparison to the problem size. As demonstrated by Table 2.4a, however, it can not make effective use of more than $(n+1)/2$ processors per minimization, or a total of $(n+1)m/2$ processors in the current context. Thus when $p=nm$, its efficiency is about 0.5. As demonstrated by Equation 2.5 and Tables 2.4a and 2.4c,

Table 3.3. Speedups for Algorithm 3.1 on Hartman 3 Test Problem

	Each Local Minimization Performed by:		
	Sequential	Parallel Strategy I	Parallel Strategy III
Number of Processors			
6	5.4	5.8	5.8
12	8.7	10.5	11.3
24	12.9	17.1	20.5
1000	23.9	44.0	76.3

Parallel Strategy III is worse than Parallel Strategy I when the number of processors is small, because some of the speculative gradient evaluations that it makes, at the unsuccessful trial points, are unnecessary and delay other, useful, function evaluations. For larger numbers of processors, however, it is the better strategy because the speculative gradient evaluations use otherwise idle processors and most of them are useful. Parallel Strategy III still makes considerably less than optimal use of nm or $2nm$ processors, however, due to the wasted speculative gradient evaluations and the differing lengths of the local minimizations, which reduces the number of processors utilized towards the end of the local minimization phase. None of these strategies can utilize more than $(n+1)m$ processors.

One way to further improve the efficiency of the concurrent global optimization algorithm might be to apply the parallel local minimization strategies discussed towards the end of Section 2, including those that utilize more than $n+1$ processors, to each of the local minimizations in Algorithm 3.1. Another way would be to change the concurrent global optimization algorithm itself. Assuming that the global optimization algorithm includes some sampling of the variable space and some local minimizations, it might be desirable to overlap these phases. This might allow the sampling, which parallelizes perfectly, to use more of the processors, and the local minimizations, which are harder to parallelize, to use fewer, while giving a more efficient utilization of the processors overall. It might also reduce the effect of the varying lengths of the local minimizations. We currently are investigating an algorithm that takes this approach.

Acknowledgements

My thanks to Matt Rosing for performing the computations that are summarized in Table 3.2, to Matt and Betty Eskow for their helpful discussions about the material in Section 3, and to Richard Byrd and Jerry Shultz for helpful discussions about Section 2.

4. References

P. T. Boggs, R. H. Byrd, and R. B. Schnabel (eds.) [1985], *Numerical Optimization 1984,* SIAM, Philadelphia.

R. H. Byrd, C. Dert, A. H. G. Rinnooy Kan, and R. B. Schnabel [1986], "Concurrent stochastic methods for global optimization", Technical Report CU-CS-338-86, Department of Computer Science, University of Colorado at Boulder.

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

L. C. W. Dixon [1981], "The place of parallel computation in numerical optimization I, the local problem", Technical Report No. 118, Numerical Optimisation Centre, The Hatfield Polytechnic.

R. Fletcher [1980], *Practical Method of Optimization, Vol 1, Unconstrained Optimization,* John Wiley and Sons, New York.

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

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

F. A. Lootsma [1984], "Parallel unconstrained optimization methods," Report No. 84-30, Department of Mathematics and Informatics, Technische Hogeschool Delft.

K. D. Patel [1982], "Implementation of a parallel (SIMD) modified Newton method on the ICL DAP", Technical Report No. 131, Numerical Optimisation Centre, The Hatfield Polytechnic.

A.H.G. Rinnooy Kan and G.T. Timmer [1984], "Stochastic methods for global optimization", to appear in the *American Journal of Mathematical and Management Sciences.*

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

R. B. Schnabel [1983], "Quasi-Newton methods using multiple secant equations," Technical Report CU-CS-247-83, Department of Computer Science, University of Colorado at Boulder.

R. B. Schnabel and P. Frank [1984], "Tensor methods for nonlinear equations", *SIAM Journal on Numerical Analysis* 21, pp. 815-843.

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

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.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER CU-CS-345-86	2. GOVT ACCESSION NO. N/A	3. RECIPIENT'S CATALOG NUMBER N/A
4. TITLE (and Subtitle) Concurrent Function Evaluations in Local and Global Optimization		5. TYPE OF REPORT & PERIOD COVERED
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Robert B. Schnabel		8. CONTRACT OR GRANT NUMBER(s) DAAG-29-84-K-0140
9. PERFORMING ORGANIZATION NAME AND ADDRESS		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS U. S. Army Research Office Post Office Box 12211 Research Triangle Park, NC 27709		12. REPORT DATE October 1986
		13. NUMBER OF PAGES 28
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

NA

18. SUPPLEMENTARY NOTES

The view, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

unconstrained optimization, global optimization, multiprocessor, concurrent function evaluation

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Attached

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)