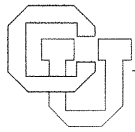
Parallel Quasi-Newton Methods For Unconstrained Optimization *

Richard H. Byrd
Robert B. Schnabel
Gerald A. Shultz

CU-CS-396-88



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

* Research supported by AFOSR grant AFOSR-85-0251, ARO contract DAAG 29-84-K-0140, NSF grants DCR-8403483 and CCR-8702403, and NSF cooperative agreement DCR-8420944.

Abstract

We discuss methods for solving the unconstrained optimization problem on parallel computers, when the number of variables is sufficiently small that quasi-Newton methods can be used. We concentrate mainly, but not exclusively, on problems where function evaluation is expensive. First we discuss ways to parallelize both the function evaluation costs and the linear algebra calculations in the standard sequential secant method, the BFGS method. Then we discuss new methods that are appropriate when there are enough processors to evaluate the function, gradient, and part but not all of the Hessian at each iteration. We develop new algorithms that utilize this information and analyze their convergence properties. We present computational experiments showing that they are superior to parallelization of either the BFGS method or Newton's method under our assumptions on the number of precessors and cost of function evaluation. Finally we discuss ways to effectively utilize the gradient values at unsuccessful trial points that are available in our parallel methods and also in some sequential software packages.

1. Introduction

This paper discusses parallel quasi-Newton methods for solving the unconstrained optimization problem,

$$\underset{x \in \mathbf{R}^n}{\text{minimize}}\ f : \mathbf{R}^n \to \mathbf{R} \ . \tag{1.1}$$

Our main emphasis is on new methods that effectively utilize multiple processors to perform multiple function and derivative evaluations simultaneously. We predominantly use these multiple function evaluations to calculate or approximate derivative values; this results in new methods that have different derivative information available than in standard sequential algorithms. Both the theoretical properties and the computational performance of these new methods are discussed. In addition, we consider the parallelization of the main linear algebra costs of such methods.

The unconstrained optimization problem (1.1) arises in many applications in science, engineering, and other areas, and is often very expensive to solve. Frequently this because the evaluation of $f(x)$ itself is expensive, often requiring many seconds or minutes on modern computers. Problems with expensive function evaluations are our main concern in this paper. It is commonly the case in such problems that analytic derivatives are not available; we concern ourselves mainly, but not exclusively with this case.

Due to the expense of many unconstrained optimization problems, there is ample incentive for trying to solve them on parallel computers. If the leading expense is the evaluation of $f(x)$ and its derivatives, then one possibility is simply to parallelize each of these evaluations. The effectiveness of this approach depends on how readily a parallel routine for $f(x)$ (and its derivatives) is available, and how fully it parallelizes the evaluation. In any case, this approach usually is outside the domain of the optimization algorithm designer. In this paper, we concentrate on the opposing case when the evaluation of $f(x)$ is assumed to be sequential, and parallelism is introduced in the optimization algorithm itself. This approach will be appropriate whenever a good parallel implementation of $f(x)$ is not available, or when the remaining costs of the optimization algorithm (such as linear algebra) are significant. In addition, on a massively parallel machine our approach might effectively be combined with parallel evaluation $f(x)$ in a multilevel parallel scheme.

Since we are interested in performing multiple evaluations of an arbitrary function $f(x)$, or its derivatives, concurrently, our parallel methods require a MIMD computer. This is a computer which can perform different calculations on different data at the same time. By contrast, an SIMD computer, which performs the same calculation on different data at the same time, will not be appropriate in general, since each evaluation of a complex function will in general require a different sequence of arithmetic operations.

Almost any kind of MIMD computer is likely to appropriate for the algorithms discussed herein. This includes both shared memory multiprocessors, or distributed memory multiprocessors such as hypercubes or networks of computers. The reason is that the granularity of the parallel operations, one or more evaluations of $f(x)$, will overwhelm any communication or synchronization overhead cost once $f(x)$ requires even a moderate number of floating point operations. This issue is discussed in more detail in Section 2.2. When n is not very large, the parallelization of the linear algebra that we discuss may be more appropriate for shared-memory multiprocessor than for distributed memory multiprocessors; this is discussed further in Section 2.3.

The methods discussed in this paper are all in the general class of quasi-Newton methods. These include secant methods, and finite difference Newton methods. On sequential computers, secant methods are generally used to solve (1.1) when function evaluation is expensive, the analytic Hessian $\nabla^2 f(x)$ is unavailable, and n is not too large. They use an approximation to the Hessian matrix that is formed from the gradient values of the iterates, and require n^2 storage and $O(n^2)$ arithmetic operations per iteration (see e.g. Fletcher [1980], Gill, Murray, Wright [1981], Dennis & Schnabel [1983]). They have been traditionally used for problems with up to about 100 variables, although with the greater storage and speed of parallel computers, they may become useful for larger dimensional problems. The finite difference Newton's method instead forms a finite difference approximation to the Hessian from function or gradient values, and requires n^2 storage and $O(n^3)$ arithmetic operations per iteration. It is generally used when the analytic Hessian is unavailable and function evaluation is inexpensive, for problems of up to 50 to 100 variables.

The remainder of this paper is concerned with constructing quasi-Newton methods that are appropriate for parallel computers. In Section 2 we discuss the parallelization of the standard sequential secant method for unconstrained optimization, the BFGS method. This topic could be considered somewhat unexciting since no new optimization algorithm is involved. But it leads to effective use of parallel

processors, and may be all that is needed in many situations. Furthermore, it leads to the consideration of two important techniques. The first is the speculative evaluation of function values introduced by Schnabel [1987], which is also the basis for the new optimization algorithms discussed in Sections 3 and 4. The second is the effective parallelization of linear algebra calculations. We compare various methods for organizing these calculations, including the method of Han [1986] (derived in a different way) and discuss which method is best in which MIMD environments. In this section we also summarize the results of some simple experiments with our parallel BFGS algorithm on a Sequent shared memory multiprocessor.

The main contributions of this paper are contained in Section 3. There we further develop a class of new methods, introduced in Byrd, Schnabel, and Shultz [1987], that evaluate the function, gradient, and part of the finite difference Hessian at each iteration. These are appropriate in two situations, both of practical interest: when the function and analytic gradient are naturally computed together on one processor and the number of processors, p, is between 2 and n, or when the gradient is approximated by finite differences and $2n+1 \leq p \leq (n^2+3n)/2$. We extend the development of the methods presented in Byrd, Schnabel, and Shultz [1987], and present both convergence analysis and computational results for what appears to the best of our new methods.

The methods discussed in Section 3 fall in between the BFGS method and a finite difference Newton's method. An important aspect of these methods is that, as we explain in Section 3, they are not generally expected to result in a speedup of p over the BFGS method on p processors. This implies that on a sequential computer, the new methods will generally be inferior to the BFGS in terms of total function and derivative evaluations required. For this reason, this class of methods has apparently not been considered prior to the start of our work on this subject. But in many practical situations on parallel computers, the new methods will be shown to be superior, in terms of time required, to either the parallelization of the BFGS discussed in Section 2 or a similar parallelization of a finite difference Newton's method. So these new methods are relevant as long as overall speed, and not just throughput measured in problems solved per processor, is of interest.

In Section 4 we discuss how a different, more minor improvement can be made to the parallel BFGS method of Section 2. It involves utilizing gradient values at failed trial points, which are available in the parallel algorithm, to reduce the total number of iterations required by the algorithm. Computational

results show that some savings are possible. In some sequential codes, this gradient information is also available and the same savings are possible. Finally, Section 5 summarizes our results and discusses interesting directions for future research.

2. Parallelizing the Standard BFGS Method

2.1. The Sequential BFGS Method

Perhaps the most commonly used method for solving multivariate unconstrained optimization problems is the BFGS method. It is intended for problems where the number of variables is small enough that the cost of storing an $n \times n$ matrix, and performing $O(n^2)$ arithmetic operations per iteration, is acceptable; otherwise conjugate direction methods (see e.g. Gill, Murray, and Wright [1981] or Dennis and Schnabel [1987]) are used. Generally the largest n for which the BFGS method is applied has been around 100, but this limit may rise with the availability of faster (sequential or parallel) computers with larger memories. The BFGS method is most appropriate when, in addition, $f(x)$ is expensive and second derivatives are unavailable. Otherwise Newton's method or a finite difference Newton's method may be faster, although the BFGS is still often used in practice.

A high level description of a BFGS algorithm is given in Algorithm 2.1. This description hides many details of the method, for example the calculations in the line search. But it is sufficient to indicate the important characteristics and costs of the method, which in turn motivate the parallel methods discussed in the remainder of this paper. For a more detailed description of the BFGS algorithm, see for example Dennis and Schnabel [1983].

There are two main categories of expense in the BFGS algorithm : function and derivative evaluation, and linear algebra calculations. The function evaluations occur in the line search, where f is evaluated at one or more trial points $x_k + \lambda_k d_k$ (with different values of λ_k), culminating in a successful point that becomes x_{k+1}. Computational experience has shown that hardly more should be required of the successful point than that it decrease the value of f. In this case, the first trial point is often successful, and

---

Algorithm 2.1 -- BFGS Method for Unconstrained Optimization

Given x_0 , $f(x_0)$, $g_0 = \nabla f(x_0)$ (or finite difference approximation), $B_0 \in \mathbf{R}^{n \times n}$ positive definite (e.g. $B_0 = I$)

At iteration k :

{ calculate search direction }
solve $B_k d_k = -g_k$ for d_k { d_k is search direction }

{ line search }
repeat
choose value of steplength λ_k
evaluate $f(x_k + \lambda_k d_k)$ (and possibly $\nabla f(x_k + \lambda_k d_k)$)
until $x_k + \lambda_k d_k$ is satisfactory next iterate

$x_{k+1} := x_k + \lambda_k d_k$
evaluate $g_k = \nabla f(x_k + \lambda_k d_k)$ (or finite difference approximation) if not already evaluated during line search

decide whether to stop ; if not :

{ update Hessian approximation }
$s_k := x_{k+1} - x_k$, $y_k := g_{k+1} - g_k$

$$B_{k+1} := B_k - \frac{B_k s_k s_k^T B_k}{s^T B_k s_k} + \frac{y_k y_k^T}{s_k^T y_k} \qquad \{ \text{BFGS update} \}$$

---

rarely are more than two or three needed; an average of 1.2 - 1.5 trial points per iteration is typical for many problems. Either during or after the line search, the gradient at the successful next iterate x_{k+1} also is calculated. (Very rarely, a gradient value may be calculated at an unsuccessful trial point during the line search.)

Thus each iteration of the BFGS method generally consists of one or more function evaluations followed by one gradient calculation at the last point where the function was evaluated. Often when $f(x)$ is expensive to evaluate, no procedure is available to calculate the gradient analytically. In this case, the gradient at any point $\bar{x}$ is approximated by the finite difference formula

$$\nabla f(\bar{x})_i \doteq g_i = \frac{f(\bar{x} + \mu_i e_i) - f(\bar{x})}{\mu_i} \tag{2.1}$$

where e_i is the i^{th} unit vector and μ_i usually is set to $macheps^{1/2} |\bar{x}_i|$. This approximation requires n evaluations of $f(x)$ in addition to $f(\bar{x})$. So when finite difference gradients are used, each iteration of the

BFGS method usually requires $n+1$ or $n+2$ evaluations of $f(x)$.

Now we return to the linear algebra costs of the BFGS method. There are two main linear algebra calculations in Algorithm 2.1, the calculation of the step direction d_k, and the calculation of the new Hessian approximation B_{k+1}. The calculation of B_{k+1} involves a rank two update to B_k that clearly requires a small multiple of n^2 operations. The calculation of d_k appears to require the solution of a system of linear equations, and hence $O(n^3)$ operations, at each iteration, but by either updating a factorization of B_k or by directly updating the inverses of B_k, this cost can be reduced to a small multiple of n^2 operations. These techniques as well as their consequences for parallel computation are discussed in Section 2.3. It will be seen that the entire linear algebra cost of the BFGS method can be limited to $2n^2 + O(n)$ multiplications, and the same number of additions and subtractions, per iteration. An important feature of the BFGS method is that each Hessian approximation B_k is symmetric and positive definite, so that each direction d_k is guaranteed to be a descent direction.

Since the linear algebra costs of the BFGS method are so small, it is easy for function and derivative evaluation to be the dominant cost. For example, if gradients are being approximated by finite differences and if each function evaluation requires at least $20n$ multiplications and additions, then function and derivative evaluation will account for at least 90% of the total cost of the method on a sequential computer. (We are disregarding some other overhead costs, such as operating system costs, but for even moderate n our estimates are accurate.) In fact many real problems we have encountered have function evaluations that are far more expensive than this. Therefore in this paper we concentrate on parallel approaches that reduce the cost of function and derivative evaluation by calculating multiple function or derivative values concurrently. Section 2.2 discusses concurrent function and derivative evaluation in the context of the standard BFGS method, while Sections 3 and 4 discuss new optimization methods that utilize concurrent function and derivative evaluation.

It is still necessary to consider parallelization of the linear algebra calculations in the BFGS method, for several reasons. First, if these calculation are performed sequentially, they may become a bottleneck on a parallel computer. Second, there are some problems where n is rather large and function evaluation rather cheap so that the linear algebra costs may be significant. We consider the parallelization of the linear algebra calculations of the BFGS method in Section 2.3. While we don't explicitly consider the

parallelization of the linear algebra calculations of our new methods of Sections 3 and 4, the techniques discussed in Section 2.2 are directly applicable to these new methods as well.

2.2 Concurrent Function Evaluation in the Standard BFGS Method

In most problems where $f(x)$ is expensive to evaluate, the gradient is not available analytically. Instead it is calculated by the finite difference approximation (2.1). We restrict ourselves to this case in this section. The new approaches of Section 3 will be seen to apply both to problems where $\nabla f(x)$ is calculated analytically and where it is approximated by finite differences.

The most obvious source of parallelism in an algorithm that uses finite difference gradients is to perform the n extra evaluations of $f(x)$ required by (2.1) concurrently. If p processors are available, this requires $\lceil n/p \rceil$ *concurrent function evaluation steps*, steps where each processor performs at most one function evaluation. The drawback to this approach is that during the evaluation of $f(x)$ in the line search, the remaining $p-1$ processors are idle. If $p \ll n$, this is unimportant since each finite difference gradient requires many concurrent function evaluation steps while each function evaluation requires just one or two, so this simple approach gives good speedups for expensive f. If $p=n$, however, then the maximum speedup that can be obtained on problems with expensive function evaluation from parallelizing only the finite difference gradient calculation is about $n/2$, or at most half of optimal. This is because both function and gradient evaluations require one concurrent function evaluation step, $n-1$ processors are idle during each function evaluation, and there are at least as many function evaluations as gradient evaluations. A more precise analysis is given below. If $p > n$, $p-n$ processors are not utilized by this approach.

An improvement on the above strategy was suggested by Schnabel [1987]. It simply is, while one processor is evaluating $f(x_k+\lambda_k d_k)$ during the line search, to utilize the remaining $p-1$ processors to evaluate $\max\{p-1,n\}$ components of $\nabla f(x_k+\lambda_k d_k)$. We refer to this as a *speculative* evaluation of (part of) the finite difference gradient. If $x_k+\lambda_k d_k$ is accepted as the next iterate, as it is most of the time, then this gradient information is required by Algorithm 2.1 and only $n+1-p$ function evaluations remain for the finite difference gradient, none if $p \geq n+1$. If $x_k+\lambda_k d_k$ is not accepted, then this gradient information is not

used by Algorithm 2.1, but nothing has been lost in comparison to the approach described in the previous paragraph. Furthermore we show in Section 4 how to make some good use of the gradient information at failed trial points by changing the optimization algorithm.

If the average number of trial points per iteration in the line search is t, then the strategy of concurrent, speculative finite difference gradient evaluation requires

$$\left\lceil \frac{n+1}{p} \right\rceil + \delta \tag{2.2}$$

concurrent function evaluation steps per iteration, as opposed to $n+1+\delta$ steps for the sequential method, and

$$\left\lceil \frac{n}{p} \right\rceil + 1 + \delta \tag{2.3}$$

for the first parallel method that parallelizes the finite difference gradient evaluation only. Thus when function evaluation is the dominant cost, the new method will make nearly optimal utilization of $n+1$ or fewer processors as long as $\delta \ll 1$. (Recall that this is usually the case in practice.) The main cases which are not addressed satisfactorily by this approach are situations when p is greater than $n+1$, or when the gradient is calculated analytically. These are addressed in Section 3.

We have run experiments on a Sequent shared memory multiprocessor to show that the speedups predicated by the above discussion are achieved in practice. We compared a parallel BFGS method utilizing speculative, concurrent finite difference gradient evaluations and the parallel linear algebra discussed in Section 2.3 to a sequential BFGS algorithm. We chose 4 standard test problems with n=40, the extended versions of Rosenbrock's function, Powell's singular function, Broyden's tridiagonal function, and the variably dimensioned function (see Moré, Garbow, and Hillstrom [1981]), with one modification : we introduced a meaningless loop into each function evaluation so that the total cost of the function evaluation would be about $20n$ flops, meaning that function evaluation would account for about 90% of the cost of the entire optimization algorithm. On the 6 processors available to us, the timed speedups ranged from 5.7 to 6.0. These numbers were in close agreement to those predicted by equation (2.2), and underscore the point that if $p<n$, function evaluation is expensive, and finite difference gradients are used, then it is easy to parallelize the BFGS algorithm almost fully.

Finally, we note a different, related approach that has been suggested by several authors (Dixon [1981], Dixon and Patel [1982], Patel [1982], Lootsma [1984], van Laarhoven [1985]) for utilizing multiple processors during the line search in the BFGS (or Newton's) method. It is to utilize the additional $p-1$ processors that are available while $f(x_k+\lambda_k d_k)$ is being evaluated to evaluate $f(x)$ at other trial points, in the direction d_k from x_k and perhaps in other directions as well. As opposed to the strategies discussed above, this strategy changes the optimization algorithm and, hopefully, sometimes results in a better next iterate and thus a smaller total number of iterations being needed to solve the optimization problem.

An interesting question is whether this approach is superior to the approach discussed above, namely using the extra processors to perform a speculative evaluation of part of the gradient during the line search. Note that the cost per iteration of the "extra line search points" (ELSP) approach, assuming that finite different gradients are evaluated concurrently, is given by (2.3). Thus from (2.2), we see for example that if $p \geq n+1$ and if $\delta = \delta_{BFGS} \geq 0$ for the BFGS method and $\delta=0$ (the best case) for the ELSP approach, then the ELSP approach is superior to the speculative gradient method if and only if it requires no more than $(1+\delta_{BFGS})/2$ times as many iterations as the BFGS method. Thus, if δ_{BFGS} is close to 0, the ELSP method would have to reduce the iteration count of the BFGS by almost 50% to be superior to it. We doubt that this reduction is likely in general, but would be interested in computational results that address this issue. We note finally that if a method using the ELSP approach could reduce the iteration count of the BFGS by a factor of 2 by always considering n points in the line search, then this would in fact be a better sequential algorithm than the standard BFGS as well.

2.3 Parallelizing the Linear Algebra Calculations in the BFGS Method

Aside from function and derivative evaluations, the dominant costs in the BFGS method are the rank two update of B_k and the calculation of the search direction d_k that are performed at each iteration. As mentioned before, these require at least $O(n^2)$ arithmetic operations. All the other calculations in the algorithm require at most $O(n)$ operations.

It is most convenient to think of the update and search direction calculation as being a pair performed in that order, i.e. update B_k to B_{k+1}, then calculate d_{k+1}. There are several different ways to organize these calculations. First, either the sequence of matrices $\{B_k\}$ or the sequence of inverses of these matrices $\{B_k^{-1}\}$ may be kept. Sequencing B_k^{-1} is reasonable because, from the Sherman-Morrison-Woodbury formula, if B_{k+1} is a rank two update of B_k, then B_{k+1}^{-1} is a rank two update of B_k^{-1}. An advantage of sequencing the inverses is that the calculation of the search directions d_k becomes simple and cheap.

In addition, no matter whether B_k or its inverse is kept, the approximation can be kept either as the symmetric and positive definite matrix B_k or B_k^{-1}, or as a factorization of this matrix. If the factorization is kept then it can be updated directly into the factorization of the next approximation. The general approach of updating factorizations was introduced by Gill, Golub, Murray, and Saunders [1974], while the special form used for the BFGS was introduced by Goldfarb [1976].

These approaches to the linear algebra calculations of the BFGS method are summarized in Table 2.1. For each approach, Table 2.1 shows the basic operations that are involved, and their cost in multiplications. (The number of additions and subtractions is the same as the number of multiplications, or nearly so, in each case.) The upper-left variant is the most straightforward and includes a Cholesky factorization at each iteration; it is the only variant that requires $O(n^3)$ operations. The upper-right variant is the sequencing of Cholesky factorizations as derived by Goldfarb [1976]. It involves a rank one update to the Cholesky factor L_k of B_k followed by a sequence of Given's rotations that reduce this updated matrix J_{k+1} to a new lower triangular matrix L_{k+1} that is the Cholesky factor of B_{k+1} (see Dennis and Schnabel [1983] for details). A straightforward implementation requires $6n^2$ operations but Goldfarb showed that this can be reduced to $2.5n^2$ by storing some additional vectors.

The lower-left variant results from the application of the inverse form of the BFGS update,

$$B_{k+1}^{-1} = B_k^{-1} + \frac{(s_k - B_k^{-1}y_k)\,s_k^T + s_k\,(s_k - B_k^{-1}y_k)^T}{y_k^T s_k} - \frac{(s_k - B_k^{-1}y_k)^T y_k\; s_k s_k^T}{(y_k^T s_k)^2} \tag{2.4}$$

followed by the multiplication of B_{k+1}^{-1} by g_{k+1} to calculate d_{k+1}. If the calculations are organized as follows

$$t = B_{k+1}^{-1}\, g_{k+1}$$

$$z = s_k - t + d_k$$

Table 2.1 -- Four Possible Implementations of the Linear Algebra Calculations :

$B_{k+1} = B_k$ + rank-two-matrix
solve $B_{k+1}\, d_{k+1} = -g_{k+1}$ for d_{k+1}

	Matrix Stored Unfactored	Matrix Stored Factored
Direct (B_k) Update	(B_k stored, updated to B_{k+1}) $B_{k+1} = B_k$ + rank-two Cholesky factor B_{k+1} 2 triangular solves to find d_{k+1} $\frac{n^3}{6} + 2n^2$	(L_k lower triangular stored, for which $B_k = L_k\, L_k^T$, updated to L_{k+1} lower triangular for which $B_{k+1} = L_{k+1}\, L_{k+1}^T$) $J_{k+1} = L_k$ + rank-one $J_{k+1} = Q_{k+1}\, L_{k+1}$ by Givens rotations 2 triangular solves to find d_{k+1} $6n^2$ $(2.5n^2)$
Inverse (B_k^{-1}) Update	(B_k^{-1} stored, updated to B_{k+1}^{-1}) $B_{k+1}^{-1} = B_k^{-1}$ + rank-two Matrix-vector multiply to find d_{k+1} $2n^2$	(M_k stored for which $B_k^{-1} = M_k\, M_k^T$, updated to M_{k+1} for which $B_{k+1}^{-1} = M_{k+1} M_{k+1}^T$) $M_{k+1} = M_k$ + rank-one 2 Matrix-vector multiples to find d_{k+1} $4n^2$

$$\gamma = s_k{}^T y_k \ , \ \delta = z^T y_k$$

$$z = z + \frac{\delta}{2\gamma} s_k \qquad (2.5)$$

$$B_{k+1}^{-1} = B_k^{-1} + z\, s_k^T + s_k\, z^T$$

$$\gamma = s_k^T g_{k+1} \ , \ \delta = z^T g_{k+1}$$

$$d_{k+1} = t + \gamma z + \delta s_k$$

then only one matrix vector multiplication, and a rank-two update of a symmetric matrix, are required, each needing n^2 multiplications as long as only the lower (or upper) triangle of each B_k^{-1} is stored.

The lower-right variant is to keep a factorization $M_k M_k^T$ of B_k^{-1}, and update M_k by the rank-one formula for the BFGS update of the factorization of the inverse to the M_{k+1} for which $M_{k+1} M_{k+1}^T = B_{k+1}^{-1}$. In

this case there is no advantage in keeping the factors triangular since the cost of doing this would outweigh the advantage in calculating d_{k+1}. This implementation of the BFGS has received less attention than the others, although it has been discussed by several authors including Brodlie, Gourlay, and Greenstadt [1973], Davidon [1975], and Powell [1987]. Recently Han [1986] derived the same implementation of the BFGS linear algebra from a rather different viewpoint.

In exact arithmetic, these four variants of the BFGS method produce identical iterates, and differ only in the number of operations required. In finite precision arithmetic, however, they may produce different iterates. Optimization folklore has long held that the unfactored inverse update may be less stable than the factored direct update. Since the inverse updates appears more attractive for parallel computation (see below), we decided to test this belief experimentally. We inserted each of the four variants of the BFGS update described in Table 2.1 into the line search BFGS method in the UNCMIN package of Schnabel, Koontz, and Weiss [1985], and tested each on the test set of Moré, Garbow, and Hillstrom [1981]. The differences in performance were negligible, averaging no more than 1-2% overall with little variation on specific problems. J. Nocedal [1987, private communication] has obtained similar results on a broader set of test problems that included some specifically designed to give the inverse variant difficulties. L. Grandinetti [1978] reports similar results.

Thus we consider any of the variants in Table 2.1 as valid points of departure for the construction of parallel BFGS methods. It is possible that the difficulty with the inverse updates may be greater for the DFP update, where there may be a larger tendency to produce numerically indefinite inverse approximations, and that this may have been the basis of the folklore about inverse updates that was then extended to include the BFGS. This possibility was pointed out to us by J. Moré [1987].

Now we consider the implementation of the linear algebra of the BFGS method on parallel computers. The unfactored direct method remains least attractive alternative on parallel computers because of its high operation count, coupled with the fact that we will see that some of the cheaper methods parallelize excellently. The factored direct method also appears to be less attractive than the two inverse methods. This is because any straightforward implementation of this approach requires a sequence of $O(n)$ vector-vector operations, such as Given's rotations. This leads to a considerably higher amount of synchronization and communication than in the inverse methods, and also does not lead directly to matrix-vector

operations, which often lead to more efficient utilization of parallel computers.

On the other hand, both of the inverse approaches seem to lend themselves excellently to implementation on either shared or local memory multiprocessors. Both consist only of matrix-vector multiplications and rank-one updates, which parallelize fully and can be implemented as block operations. On a shared memory multiprocessor with $p \leq n$ processors, we would expect the unfactored direct approach to require time proportional to $2n^2/p$, and the factored inverse approach to require time proportional to $4n^2/p$. Other considerations, such as caching, seem similar for the two approaches. It is possible that the rank one update of a triangular matrix, required by the unfactored inverse approach, would not parallelize quite as well as the other operations in conjunction with some caching policies.

On a local memory multiprocessor, it appears that, in order to avoid excessive communication, the unfactored inverse approach would need to store and update the full matrix B_k^{-1} (partitioned by rows) rather than just the upper or lower triangle. This raises the total cost of the method to $3n^2$ operations which narrows the gap between it and the factored inverse approach. Again the arithmetic operations should parallelize fully for both approaches. In addition, both approaches appear to require the same amount of information to be communicated per iteration, although the factored method seems to only require one synchronization point whereas the unfactored method seems to require two.

From the above discussion, we would expect the unfactored inverse approach to be the best way to implement the linear algebra operations of the BFGS method on a shared memory multiprocessor. It would also appear to be the best approach for a local memory multiprocessor, but it should be tested against the factored inverse approach. On a shared memory multiprocessor, the synchronization costs are small and the parallel BFGS should be efficient for almost any values of n and p. For the parallelization of the BFGS to be efficient on a local memory multiprocessor, the number of floating point operations per processor per iteration, about $3n^2/p$, must significantly exceed the cost of sending either one or two messages that contain a total of about $3n$ floating point numbers.

The parallel BFGS code mentioned at the end of Section 2.2 uses a parallel version of the unfactored inverse approach. To test how well all the linear algebra calculations are parallelized, we ran this code on a Sequent shared memory multiprocessor on the cheapest possible objective function, $f(x) = x^T x$. Thus the linear algebra calculations are the dominant cost. We also parallelized most of the $O(n)$ computations,

although inner products were left sequential. We found that the speedup on 6 processors was only about 3.7 for $n = 40$, and 4.3 for $n = 100$. These results plus our results using fewer processors indicated that approximately 12% of the code remained sequential for $n = 40$, while approximately 8% remained sequential when $n = 100$. This indicates the importance of parallelizing all the $O(n)$ calculations, as well as the $O(n^2)$ calculations, in a parallel implementation of the BFGS method.

3. Parallel Methods That Use Part Of The Finite Difference Hessian

3.1 Approaches to Using Partial Hessian Information

We now consider a class of methods that use parallel processors to evaluate part, but not all, of the finite difference Hessian matrix $\nabla^2 f(x)$ along with the function and gradient at each trial point. Our orientation is towards problems where function and derivative evaluation is the dominant cost. As discussed previously, this is the case for many practical problems.

The approaches that we discuss fall in between the BFGS method, which uses only the function and gradient at each trial point, and Newton's method, which uses the function, gradient, and Hessian. Implicit in this statement are two assumptions. First, that if we have enough processors to evaluate the function, gradient, and Hessian in one concurrent function evaluation step, then we will do this and use a modern Newton's method based algorithm (see e.g. Moré and Sorensen [1983]). Second, that if we do not have enough processors to do this, then we will probably not want to use extra concurrent function evaluation steps to evaluate the full Hessian at each iteration. This second assumption is motivated by considerable computational experience (see e.g. Schnabel, Koontz, and Weiss [1985]) that shows that the iterations saved by using a finite difference Newton's method algorithm rather than the BFGS method usually do not offset the extra cost per iteration in function evaluations. The results of Section 3.3 will validate this assumption.

Thus we consider the approach of partial Hessian evaluation whenever there are not enough processors to evaluate the function, gradient, and Hessian in one concurrent function evaluation step, but more than enough to evaluate just the function and gradient. This occurs in two distinct situations, both of

practical interest. The first is when the gradient is evaluated by finite differences and the number of processors is greater than $n+1$ but less than $(n^2+3n+2)/2$. In this case, there are more than enough processors to evaluate the function and finite difference gradient concurrently at each trial point, but not enough to evaluate the function, finite difference gradient, and full finite difference Hessian. For example, on a 64 node hypercube, this is the case whenever $n \in [10,63]$. The second scenario we consider is when the analytic gradient is readily computed along with the function value, so that it is most convenient to computer both on one processor, but the analytic Hessian is not available. This is the case in a reasonable number of practical problems, for instance many optimal control problems. In this case, if the number of processors is between 2 and n, we again have more processors than are needed for just the function and gradient, but not enough for the full finite difference Hessian (which requires n additional gradient values) as well.

In either of these cases, the methods of this section use the excess processors to compute as large a portion of the finite difference Hessian as possible at each iteration. An interesting aspect of these algorithms is that while they will be seen to be worthwhile on parallel computers whenever the partial Hessian evaluation uses otherwise unutilized processors, or if the goal is absolute speed (rather than speed per processor), they are not in general the most efficient methods on sequential computers. Probably for this reason, they have apparently not been considered prior to our investigations.

Byrd, Schnabel, and Shultz [1987] proposed a variety of approaches for utilizing partial Hessian information, and examined some of their computational and theoretical properties. The general approach that they found to be best is outlined in Algorithm 3.1. The remainder of Section 3.1 continues the development of this approach. In Sections 3.2 and 3.3 we present new theoretical and computational results about this type of method.

Algorithm 3.1 differs from the standard BFGS method, Algorithm 2.1, in several ways. First, the speculative gradient evaluation discussed in Section 2.2 is performed at each trial point in the line search. Second, speculative evaluation of some portion of the Hessian also is performed at each trial point in the line search. Third, this partial Hessian information is incorporated into the Hessian approximation at each iteration, following the standard BFGS update. We now briefly discuss the motivation for these steps and some of the alternatives considered in Byrd, Schnabel, and Shultz [1987]. We also introduce some new aspects of these steps.

Algorithm 3.1 -- Quasi-Newton Method for Unconstrained Optimization Using Speculative Partial Hessian Evaluation

Given x_0 , $f(x_0)$, $g_0 = \nabla f(x_0)$ (or finite difference approximation), $B_0 \in \mathbf{R}^{n \times n}$ positive definite (e.g. $B_0 = I$), $q \in [1, n-1]$

At iteration k :

{ calculate search direction }
solve $B_k d_k = -g_k$ for d_k { d_k is search direction }

{ line search }
choose set of q linearly independent vectors $u_1, \cdots, u_q$

repeat
choose value of steplength λ_k
evaluate $f(x_k + \lambda_k d_k)$, $\nabla f(x_k + \lambda_k d_k)$ (or finite difference approximation), and finite difference approximation to $\nabla^2 f(x_k + \lambda_k d_k)\, u_i$ for each $i \in [1, q]$
until $x_k + \lambda_k d_k$ is satisfactory next iterate

$x_{k+1} := x_k + \lambda_k d_k$

decide whether to stop ; if not :

{ update Hessian approximation }
$s_k := x_{k+1} - x_k$, $y_k := g_{k+1} - g_k$

$$\bar{B}_{k+1} := B_k - \frac{B_k s_k s_k^T B_k}{s^T B_k s_k} + \frac{y_k y_k^T}{s_k^T y_k} \qquad \{ \text{BFGS update} \}$$

B_{k+1} := update of $\bar{B}_{k+1}$ based on the finite difference information $\nabla^2 f(x_{k+1})\, u_i$, $i = 1, \cdots, q$

The partial Hessian information that is approximated in Algorithm 3.1 is $\nabla^2 f(x)\, u_i$, $i = 1, \cdots, q$. Byrd, Schnabel, and Shultz considered two choices of the vectors u_i that are selected at each iteration : a set of q unit directions, or a set of q conjugate directions. They found that using conjugate directions led to no significant advantage in the context of Algorithm 3.1, and that it caused a considerable extra linear algebra cost. Therefore, we only consider the use of unit directions below. That is, at each iteration we select $\{u_i\}$ by

$$\text{choose a set } \Gamma_k \text{ of distinct integers between 1 and } n$$

$$u_i = e_{\gamma_i}, \text{ where } \gamma_i \text{ is the } i^{th} \text{ member of } \Gamma_k \ . \qquad (3.1)$$

This means that our algorithm approximates q columns of $\nabla^2 f(x)$, whose indices are given by Γ_k, at each iteration. In our computational implementation, we choose the sequence of sets Γ_k to cycle through the

indices 1 to n.

If the function and gradient are evaluated analytically together on one processor, then column i of the Hessian at $\bar{x} = x_k + \lambda_k d_k$ can be approximated by calculating $\nabla f(\bar{x} + \mu_i e_i)$ where $\mu_i = macheps^{1/2} |\bar{x}_i|$, and then setting

$$\nabla^2 f(\bar{x}) e_i = h_i = \frac{\nabla f(\bar{x} + \mu_i e_i) - \nabla f(\bar{x})}{\mu_i} \tag{3.2}$$

Thus if $p \in [2, n]$ processors are available, then q will be set to $p-1$ and q columns of the Hessian will be evaluated using q additional gradient evaluations.

If the gradient is not available analytically, then the only way to approximate the gradient or Hessian is from finite differences involving function values. Let $\bar{\Gamma}_k = \{j \mid j \in \{1,2,\cdots,n\},\ j \notin \Gamma_k\}$. A new, efficient way to approximate the gradient and q columns of the Hessian at $\bar{x}$ is to use the formulas

$$\nabla^2 f(\bar{x})_{ij} \cong (h_j)_i = \frac{f(\bar{x} + \mu_i e_i + \alpha_j e_j) - f(\bar{x} + \mu_i e_i) - f(\bar{x} + \alpha_j e_j) + f(\bar{x})}{\mu_i \alpha_j} \tag{3.3a}$$

$$\nabla f(\bar{x})_i \cong h_i = \frac{f(\bar{x} + \mu_i e_i) - f(\bar{x})}{\mu_i} \tag{3.3b}$$

for $i \in \bar{\Gamma}_k$, $j \in \Gamma_k$, where $\mu_i = macheps^{1/2} |\bar{x}_i|$ and $\alpha_j = macheps^{1/4} |\bar{x}_j|$,

$$\nabla^2 f(\bar{x})_{ij} \cong (h_j)_i = \frac{f(\bar{x} + \beta_i e_i + \beta_j e_j) - f(\bar{x} + \beta_i e_i) - f(\bar{x} + \beta_j e_j) + f(\bar{x})}{\beta_i \beta_j} \tag{3.3c}$$

$$\nabla^2 f(\bar{x})_{ii} \cong (h_i)_i = \frac{f(\bar{x} + \beta_i e_i) - 2f(\bar{x}) + f(\bar{x} - \beta_i e_i)}{\beta_i^2} \tag{3.3d}$$

for $i, j \in \Gamma_k$, $i \equiv j$, where $\beta_i = macheps^{1/3} |\bar{x}_i|$,

$$\nabla f(\bar{x})_i \cong h_i = \frac{f(\bar{x} + \beta_i e_i) - f(\bar{x} - \beta_i e_i)}{2\beta_i} \tag{3.3e}$$

for $i \in \Gamma_k$, with the same β_i. Using these formulas, we can approximate the function, gradient and q columns of the Hessian using $(n+1-(\frac{q}{2}))(q+1)$ function evaluations. Thus if p processors are available, we will choose q to the the largest integer for which $(n+1-(\frac{q}{2}))(q+1) \leq p$. A side benefit of the above formulas is that for each $i \in \Gamma_k$, the i^{th} component of the gradient is approximated by central differences and hence is more accurate than the value given by the standard forward difference approximation (2.1), at no additional cost in function evaluations.

The partial Hessian information is incorporated into the new Hessian approximation after the new iterate x_{k+1} is selected. At this point Algorithm 3.1 potentially has $q+1$ new pieces of information to incorporate into the Hessian approximation : the standard secant equation

$$B_{k+1} s_k = y_k \ , \tag{3.4}$$

and the q finite difference values

$$B_{k+1} u_i = z_i \ , \quad i=1,..., q \tag{3.5}$$

where $u_i = e_{\gamma_i}$ and z_i is the finite difference approximation to column γ_i of $\nabla^2 f(x_{k+1})$. We incorporate the standard secant equation (3.4) first, and then the finite difference information (3.5). This order seems reasonable because the standard secant equation gives, in some sense, information about the Hessian value in between x_k and x_{k+1}, while the finite difference information is at x_{k+1} and hence is the most current information. Updating in this order means that the standard secant equation may not hold at the ultimate value of B_{k+1}, but in Section 3.2 we show that q-superlinear convergence still is retained. Byrd, Schnabel, and Shultz [1987] also considered omitting, or only temporarily using, the standard secant equation (3.4), but their computational results indicated that it is preferable to include it. This is the only possibility considered in this paper.

First we incorporate the standard secant equation (3.4) using the standard BFGS update. Then there are various ways to incorporate the finite difference information (3.5). Byrd, Schnabel, and Shultz [1987] show that using the PSB update is simply equivalent to overwriting the corresponding row and column with the finite difference information. However their computational results show that using the BFGS update may lead to a slightly more efficient algorithm, and it has the advantage of generating positive definite Hessian approximations. So we will use BFGS updates to incorporate the finite difference Hessian information. Byrd, Schnabel, and Shultz [1987] only consider in detail the case $q = 1$; now we consider how to incorporate (3.5) when $q>1$.

We have considered two ways to incorporate the partial finite difference Hessian information (3.5) by BFGS updates. The first is to perform a sequence of q standard BFGS updates, i.e.

$$\bar{B}_{k+1,i+1} = \begin{cases} \bar{B}_{k+1,i} - \dfrac{\bar{B}_{k+1,i}\, u_i\, u_i^T \bar{B}_{k+1,i}}{u_i^T \bar{B}_{k+1,i}\, u_i} + \dfrac{z_i\, z_i^T}{u_i^T z_i} & \text{if } u_i^T z_i > 0 \\ \bar{B}_{k+1,i} & \text{otherwise} \end{cases} \tag{3.6}$$

for $i=1, \cdots, q$, where $\bar{B}_{k+1,1} = \bar{B}_{k+1}$ and $B_{k+1} = \bar{B}_{k+1,q+1}$. This procedure has the advantage of simplicity, but the possible disadvantage that B_{k+1} will, in general, only obey the last finite difference equation of (3.5) exactly.

The second alternative is to use multiple secant updates (Schnabel [1983]). Let $U \in \mathbf{R}^{n\times q}$ have as its columns u_i, $i=1,..., q$, and let $Z \in \mathbf{R}^{n\times q}$ have as its columns z_i, $i=1,..., q$. If $U^T Z \cong U^T \nabla^2 f(x_{k+1})$ is positive definite, we use the multiple (rank $2q$) BFGS update

$$B_{k+1} = \bar{B}_{k+1} - \bar{B}_{k+1} U_k \,(U_k^T \bar{B}_{k+1} U_k)^{-1}\, U_k^T \bar{B}_{k+1} + Z_k\,(U_k^T Z_k)^{-1} Z_k^T \tag{3.7a}$$

This update causes B_{k+1} to satisfy all q equations in (3.5), and to be positive definite given that $\bar{B}_{k+1}$ is positive definite. If $V_k = \nabla^2 f(x_{k+1}) U_k$ exactly then the matrix $U_k^T Z_k$ is symmetric. However, if we use finite difference approximations for V_k the discretization error can cause that matrix to not be symmetric. Therefore, when using finite differences, we replace $U_k^T V_k$ with $\frac{1}{2}(U_k^T V_k + V_k^T U_k)$ in (3.7a).

If $U^T Z$ is not positive definite, we use a sequence of smaller multiple secant updates to partially enforce (3.5). First we select the subset PD of Γ_k consisting of indices i for which the equations of (3.5) are consistent with positive definiteness, i.e.

$$PD = \{i \mid i \in [1, q] \text{ and } u_i^T z_i > 0\}\ .$$

Then we use a heuristic to select a maximal subset PD_1 of PD for which $U_1^T Z_1$ is positive definite, where U_1 has as its columns u_i for all $i \in PD_1$, and Z_1 has as its columns z_i for all $i \in PD_1$. Then we similarly select a subset PD_2 containing some or all of the remaining members of PD, for which $U_2^T Z_2$ is positive definite, where U_2 and Z_2 are defined similarly. If any columns remain, we then select similar subsets $PD_3, \ldots, PD_m$, until each $i \in PD$ is in exactly one subset, and each $U_j^T Z_j$ is positive definite. Then we use the multiple BFGS formula (3.7a) to incorporate, in order, each of the equations $B_{k+1} U_j = Z_j$, for j going from m down to 1, choosing the backward order so that the maximal subset is incorporated last. That is, we perform the updates,

$$\bar{B}_{k+1,i-1} = \bar{B}_{k+1,i} - \bar{B}_{k+1,i} U_i \,(U_i^T \bar{B}_{k+1,i} U_i)^{-1}\, U_i^T \bar{B}_{k+1,i} + Z_i\,(U_i^T Z_i)^{-1} Z_i^T\ ,\quad i=m \text{ down to } 1 \tag{3.7b}$$

where $\bar{B}_{k+1,m} = \bar{B}_{k+1}$ and $B_{k+1} = \bar{B}_{k+1,0}$. In the computational implementation, we replace the criterion

$u_i^T z_i > 0$, which we have used above for simplicity of exposition, with the criterion $u_i^T z_i > macheps^{1/2} \| u_i \|_2 \| z_i \|_2$.

We have tested algorithms both the first alternative (3.6) and the second alternative (3.7), and noticed a slight advantage for the second, multiple secant approach. Therefore only this approach is considered in the computational results presented in Section 3.3. In performing the convergence analysis of Section 3.2, however, it turns out that the techniques we use to prove the convergence of the method using the multiple secant approach (3.7) build upon the convergence analysis of the sequential update approach (3.6). Therefore in Section 3.2 superlinear convergence of both of these methods of incorporating the partial finite difference Hessian information is proved.

3.2 Convergence Properties of Partial Hessian Methods

We now consider the question of convergence of the new methods discussed in the previous section. We are able to show that Algorithm 3.1 has the same properties of q-superlinear convergence and global convergence on uniformly convex functions that the BFGS method has. In particular we are able to establish results similar to some of those of Powell [1976] and Dennis and Moré [1974], although we will make use of machinery for analyzing secant methods developed by Byrd and Nocedal [1987]. The convergence results in this section will be proved under the following assumptions.

Assumptions 3.1.

(1) The objective function f has a Lipschitz continuous second derivative on the level set $\Omega = \{x : f(x) \leq f(x_0)\}$. Denote the Lipschitz constant by L.

(2) There are positive constants μ_1 and μ_2 such that for all $z \in R^n$ and all $x \in \Omega$

$$\mu_1 \| z \|^2 \leq z^T \nabla^2 f(x) z \leq \mu_2 \| z \|^2.$$

Note that this implies that f has a unique minimizer x_* in Ω.

(3) The line search used with Algorithm 3.1 has the property that there exist positive constants η_1 and η_2 such that at each iteration either

$$f(x_k+\lambda_k d_k) \le f(x_k)-\eta_1[\frac{\nabla f(x_k)^T d_k}{||d_k||}]^2, \tag{3.8}$$

or

$$f(x_k+\lambda_k d_k) \le f(x_k)+\eta_2 \nabla f(x_k)^T d_k \tag{3.9}$$

is satisfied.

(4) The line search has the property that if $\frac{||(B_k-\nabla^2 f(x_*))s_k||}{||s_k||}$ and $||x_k-x_*||$ are sufficiently small then the steplength $\lambda_k = 1$ will be used.

(5) The Hessian information used for the extra updates is exact. That is, $z_i = \nabla^2 f(x_{k+1})u_i$ in (3.6), and $Z = \nabla^2 f(x_{k+1})U$ in (3.7 a)

The line search assumption (3) is meant to be as general as possible. It can be shown that it is satisfied for some η_1, η_2 if λ_k is chosen by any standard procedure such as the Wolfe conditions (3.17-18), the Goldstein conditions, or any reasonable backtracking strategy. This condition is discussed in more detail by Byrd and Nocedal [1987]. Assumption (4) was shown to be satisfied by the Wolfe conditions by Dennis and Moré [1976], and similar arguments show that the Goldstein conditions and backtracking also satisfy Assumption (4).

Theorem 3.1. Consider Algorithm 3.1 with the finite difference updates made sequentially by (3.6), and suppose that Assumptions 3.1 are satisfied. Then the sequence $\{x_k\}$ that is produced converges super-linearly to the solution x_*.

Proof. In the sequential updating algorithm, the quasi-Newton approximation, B, is updated successively by BFGS along step directions s_k and finite difference directions u_i, $i = 1,...,q$. For the moment we will number the sequence of quasi-Newton matrices over the entire algorithm in order of computation without regard to the type of update made. Therefore we denote B_k by $B_{((q+1)k)}$ and $\bar{B}_{k+1,i}$ by $B_{((q+1)(k-1)+i)}$. Likewise we denote the directions s_k and u_i by a sequence $\{r_j\}$ where $s_k = r_{(q+1)k}$ and, at iteration k, $u_i = r_{(q+1)k+i}$. Each update then has the form

$$B_{(j+1)} = B_{(j)} - \frac{B_{(j)} r_j r_j^T B_{(j)}}{r_j^T B_{(j)} r_j} + \frac{w_j w_j^T}{w_j^T r_j}$$

where for each update

$$w_j = \bar{G}_j r_j. \tag{3.10}$$

For the finite difference based update $\bar{G}_j = \nabla^2 f(x_{k+1})$, and for the step update

$$\bar{G}_j = \int_0^1 \nabla^2 f(x_k + \tau s_k) s_k \, d\tau.$$

In either case by the uniform convexity in Assumption (3.1.2)

$$\frac{w_j{}^T r_j}{r_j{}^T r_j} = \frac{r_j{}^T \bar{G}_j r_j}{r_j{}^T r_j} \geq \mu_1, \tag{3.11}$$

and

$$\frac{w_j{}^T w_j}{w_j{}^T r_j} = \frac{r_j{}^T \bar{G}_j{}^2 r_j}{r_j{}^T G_j r_j} \leq \mu_2. \tag{3.12}$$

Now by Theorem 2.1 of Byrd and Nocedal [1987], if a sequence of BFGS updates is performed with (3.11) and (3.12) satisfied for each update, then for any fraction $\rho \in (0,1)$ there exist constants β_1 and β_2 such that for any positive integer m the bounds

$$\frac{r_j{}^T B_{(j)} r_j}{\| r_j \| \, \| B_{(j)} r_j \|} \geq \beta_1 \tag{3.13}$$

and

$$\frac{\| B_{(j)} r_j \|}{\| r_j \|} \leq \beta_2 \tag{3.14}$$

are satisfied for at least ρm values of j in $[1,m]$. (Note that the quantity in (3.13) is the cosine of the angle between r_j and $B_{(j)} r_j$.) Now if (3.13) is true for r_j a step direction then that implies that it is a strong descent direction. To ensure that many step directions are strong descent directions we take ρ to satisfy

$$\rho \geq \frac{q+\frac{1}{2}}{q+1}.$$

Then by the quoted result on the BFGS, in k outer iterations $(q+1)k$ updates are made, of which $\rho(q+1)k \geq (q+\frac{1}{2})k$ satisfy (3.13) and (3.14). Of these at least $(q+\frac{1}{2})k$ updates, at most qk are finite difference updates so that at least $\frac{1}{2}k$ of the step directions satisfy (3.13) and (3.14).

Now by Theorem 3.1 of Byrd and Nocedal [1987], if $\{x_k\}$ is generated by

$$x_{k+1} = x_k + s_k = x_k - \lambda_k B_k{}^{-1} \nabla f(x_k)$$

where, for each k, at least some fixed fraction of the directions satisfy

$$\frac{s_k{}^T B_k s_k}{\| s_k \| \, \| B_{(i)} s_k \|} \geq \beta_1$$

and

$$\frac{||B_k s_k||}{||s_k||} \leq \beta_2$$

and the line search satisfies Assumption 3.1.3, then $\{x_k\}$ converges to x_* r-linearly so that

$$\sum_{k=0}^{\infty} ||x_k - x_*|| < \infty. \tag{3.15}$$

To show that the convergence is superlinear, note that since the quasi-Newton matrix B is updated $q+1$ times at each point (3.15) implies that the matrices $\bar{G}_i$ in (3.10) satisfy

$$\sum_{i=0}^{\infty} ||\bar{G}_i - \nabla^2 f(x_*)|| \leq (q+1)L \sum_{k=0}^{\infty} \max[\, ||x_{k-1} - x_*||,\ ||x_k - x_*|| \,] < \infty.$$

Therefore by Theorem 3.2 of Byrd and Nocedal [1987], or alternatively by Theorem 3.4 of Dennis and Moré [1974], it follows that

$$\frac{||(B_k - \nabla^2 f(x_*))s_k||}{||s_k||} \to 0.$$

>From this fact, superlinear convergence of the sequence $\{x_k\}$ follows by Theorem 2.2 of Dennis and Moré [1974] and Assumption 3.1.4. □

Now we consider the multiple secant update (3.7). It turns out that doing the multiple update using an $n \times q$ matrix U is equivalent to a sequence of q simple updates along a set of conjugate directions spanning the column space of U.

Lemma 3.1. Consider a sequence of q standard BFGS updates (3.6) to the positive definite matrix $\bar{B}_{k+1}$ using directions $u_1, \ldots, u_q$ that are conjugate with respect to $\nabla^2 f(x_{k+1})$. Suppose that Assumption 3.1.5 is satisfied. Then the resulting matrix $\bar{B}_{k+1,q+1}$ is the same matrix as results from a multiple update of the form (3.7a) where the column space of the matrix U, is equal to the span of $\{u_1, \ldots, u_q\}$ as long as the matrix $U^T \nabla^2 f(x_{k+1}) U$ is positive definite.

Proof. First we note that the multiple update depends only on the column space of U. This is true since an $n \times q$ matrix having the same column space as U must have the form UT, where T is a nonsingular $q \times q$ matrix. If we then replace U by UT and Z by ZT in (3.7a) it is easy to see that the result is unchanged.

Therefore for the rest of the proof we assume without loss of generality that the columns of U are conjugate vectors $u_1, \ldots, u_q$. Since $Z = \nabla^2 f(x_{k+1}) U$, conjugacy with respect to $\nabla^2 f(x_{k+1})$ implies that

$$U^T Z = \text{diag}\,(u_i{}^T z_i)$$

so that the last term in the multiple update formula (3.7a) is

$$Z(U^T Z)^{-1} Z^T = \sum_{i=1}^{q} \frac{z_i z_i{}^T}{u_i{}^T z_i}. \tag{3.16}$$

If we consider sequential updating we see that the final matrix is

$$\bar{B}_{k+1,q+1} = \bar{B}_{k+1,0} - \sum_{k=0}^{q-1} \frac{\bar{B}_{k+1,i} u_i u_i{}^T \bar{B}_{k+1,i}}{u_i{}^T \bar{B}_{k+1,i} u_i} + \sum_{k=0}^{q-1} \frac{z_i z_i{}^T}{u_i{}^T z_i}.$$

Note that the last sum is equal to the last term in (3.7a). Now consider the q BFGS updates one at a time. For a given i, if $\bar{B}_{k+1,i} u_j = z_j \text{for} j \le i$ then

$$\bar{B}_{k,j+1} u_j = \bar{B}_{k+1,i} u_j - \frac{\bar{B}_{k+1,i} u_i u_i{}^T z_j}{u_i{}^T \bar{B}_{k+1,i} u_i} + \frac{z_i z_i{}^T u_j}{z_i{}^T u_i}$$

$$= \bar{B}_{k+1,i} u_i,$$

since by conjugacy $u_i{}^T z_j = u_j{}^T z_i = u_i{}^T \nabla^2 f(x_{k+1}) u_j = 0$. Therefore, since each update causes $\bar{B}_{k+1,i} u_{i-1} = z_{i-1}$, after q updates $\bar{B}_{k+1,q+1} u_i = z_i$ for $i = 1,...,q$, so that the q secant equations are satisfied just as for the multiple update.

Now consider the matrices

$$D_1 = \sum_{k=1}^{q} \frac{\bar{B}_{k+1,i} u_i u_i{}^T \bar{B}_{k+1,i}}{u_i{}^T \bar{B}_{k+1,i} u_i}$$

and

$$D_2 = \bar{B}_{k+1} U (U \bar{B}_{k+1} U)^{-1} U^T \bar{B}_{k+1}.$$

We have that

$$D_1 U = \bar{B}_{k+1} U + Z(U^T Z)^{-1} Z^T U - \bar{B}_{k+1,q+1} U$$

$$= \bar{B}_{k+1} U + Z_k - Z_k = \bar{B}_{k+1} U = D_2 U.$$

Therefore since the matrix $\bar{B}_{k+1} U$ has rank q and D_1 and D_2 are rank q matrices, it follows that both matrices have the same range, the column space of $\bar{B}_{k+1} U$. Since they are symmetric they also have the same null space. Together with the fact that $D_1 U = D_2 U$ this implies that $D_1 = D_2$. Therefore by (3.16) it follows that the resulting matrices are equal. □

Note that for this result we have not used any of the Assumptions 3.1 except the last one. Although we have stated this lemma for a complete multiple update of the form (3.7a), it is clear that the proof

applies to each of the partial multiple updates (3.7b) as long as each of the matrices $U_i^T \nabla^2 f(x_{k+1}) U_i = U_i^T Z_i$ in (3.7b) are positive definite. Thus the sequence of multiple updates (3.7b) is equivalent to a sequence of single BFGS updates using conjugate directions in the same order.

Given the equivalence result of Lemma 3.1 the convergence of the multiple update version of our algorithm follows immediately. Since the convergence analysis assumes positive definiteness of $\nabla^2 f(x)$, a complete multiple update will always be possible in Algorithm 3.1, and we need only consider the form (3.7a).

Theorem 3.2. Consider Algorithm 3.1 with the finite difference information incorporated using the multiple update (3.7a) with the matrix U having full rank at each iteration, and suppose that Assumptions 3.1 are satisfied. Then the sequence $\{x_k\}$ produced converges superlinearly to the solution x_*.

Proof. By Lemma 3.1, Algorithm 3.1 using a multiple update is equivalent to Algorithm 3.1 with a sequential update along a set of directions conjugate with respect $\nabla^2 f(x_{k+1})$ and spanning the column space of U. By Theorem 3.1 that version of Algorithm generates a sequence which converges to x_* superlinearly. □

We have thus shown that both versions of Algorithm preserve the convergence properties of the BFGS method. It is interesting to note that Theorems 3.1 and 3.2 put absolutely no conditions on the choice of U except that it have full rank at each step. Of course this is true because we are only trying to show that the extra updates do not interfere with the good properties of the BFGS. One might hope that there is some theoretical result showing that the finite difference updates actually improve the convergence behavior of the algorithm in some way, but we have not been able to find one. It is interesting to note that Byrd, Schnabel and Shultz [1987] prove that if the step update is omitted (or removed after step computation) the resulting algorithm is $\left\lceil \frac{n}{q} \right\rceil$ -step quadratically convergent, and this result depends very strongly on how the finite difference directions are chosen. However, as mentioned in Section 3.1 that method performs more poorly in numerical experiments than the method analyzed here.

3.3 Computational Performance of Partial Finite Difference Hessian Methods

We have tested the partial Hessian Algorithm 3.1 on a variety of test problems. Byrd, Schnabel, and Shultz [1987] report the results of tests for the case $q=1$ only, on a set of problems from Moré, Garbow, and Hillstrom [1981] with small values of n. Here, we report on tests of Algorithm 3.1 for the full range $q = 1$ to n. When $q>1$, we incorporate the partial finite difference Hessian information by the multiple secant procedure (3.7). The test problems considered are a combination of problems from Moré, Garbow, and Hillstrom [1981] and Conn, Gould, and Toint [1986], run with the values $n = 20$ and $n = 40$. They are listed in Table A1 in the appendix. The standard starting point was used for all problems except #15, where $(-0.5, 0.5, \cdots, -0.5, 0.5)$ was used because our BFGS algorithm overflowed from the standard start point $(-1, 1, \cdots, -1, 1)$.

The implementation of Algorithm 3.1 that we tested was obtained by modifying the BFGS, line search algorithm in the UNCMIN unconstrained optimization software package (Schnabel, Koontz, and Weiss [1975]) in two ways. First, at each iteration the finite difference information update (3.7) was added after the standard BFGS step update, as explained in Section 3.1. Second, the backtracking line search in UNCMIN, in which each iterate satisfies the condition

$$f(x_{k+1}) < f(x_k) + \alpha \nabla f(x_k)^T d_k \tag{3.17}$$

for $\alpha=10^{-4}$, was augmented so that each iterate also satisfies

$$\nabla f(x_{k+1})^T d_k \geq \beta \nabla f(x_k)^T d_k \tag{3.18}$$

where $\beta=0.9$ (using Algorithm A6.3.1mod in Dennis and Schnabel [1983]). With condition (3.18), a positive definite step update is always possible. The BFGS algorithm used for comparison was the same algorithm without any finite difference information. The Newton's method algorithm used for comparison was the line search, Newton's method algorithm in UNCMIN; if the Hessian is indefinite, it uses a modified Cholesky decomposition strategy described in Dennis and Schnabel [1983] to perturb the Hessian and calculate the line search direction. The standard UNCMIN stopping conditions, described in Schnabel, Koontz, and Weiss [1985], were used. The tests were run in double precision on a VAX 780.

We are primarily interested in the performance of this method on parallel computers when function evaluation is expensive. As discussed in Section 2.1, function evaluation does not have to be very expen-

sive before it swamps all other costs of the BFGS method on sequential computers. Even on a local memory multiprocessor, once each function evaluation requires several thousand floating point operations, the cost of function evaluations is likely to swamp all costs including synchronization and communication.

Thus we will evaluate Algorithm 3.1 for each value of q by simply counting the number of trial point function evaluations it requires to solve each problem, (i.e. the total number of points tried in the line searches and all the iterations, plus the starting point). If there are enough processors to evaluate the function, gradient, and q columns of the finite difference Hessian in one concurrent function evaluation step, then the number of trial point function evaluations is equivalent to the number of concurrent function evaluation steps and is indicative of the cost of Algorithm 3.1 on a parallel computer, for expensive functions. The speed of Algorithm 3.1 is then compared to the speed of the parallel BFGS method, implemented as discussed in Section 2.2. This parallel BFGS method is assumed to use speculative gradient evaluations so that the function and gradient are evaluated in one concurrent function evaluation step, but any additional processors are unused.

The raw computational results for our method for various values of q, as well as for the BFGS method and Newton's method, are given in Tables A2 and A3 in the appendix for n = 20 and 40 respectively. This data is summarized in Tables 3.1-3.3. Tables 3.1 and 3.2 give the simulated average speedups, over the parallel BFGS method, that we obtained for each value of n and several values of q. These average speedups were computed by taking all the problems solved correctly by both methods for a given value of q and n, and dividing the total number of trial points required by the BFGS method on all these problems by the total number of trial points required by the new method on all these problems. This is a reasonable measure of speedup under the assumptions that function evaluation is expensive and there are enough processors to evaluate the function, gradient, and q columns of the Hessian simultaneously. Problems not solved successfully for one or both methods are excluded when computing the speedups in Tables 3.1-3.3; we noticed no significant difference in the success rates of the various methods.

If the function and gradient are evaluated together, analytically, by one processor, then Tables 3.1-3.2 reflect the use that Algorithm 3.1 could make of from 2 to n+1 processors. If the gradient is evaluated by finite differences, then the "standard" BFGS method that is used as the comparison itself requires $n+1$ processors, and Tables 3.1-3.2 reflect the use that Algorithm 3.1 could make of from $2n+1$ to $(n^2+3n+2)/2$

Table 3.1 -- Average Speedup of Algorithm 3.1 over Parallel BFGS Method, n=20

q	1	2	3	4	5	10	20
Average Speedup	1.86	2.03	2.55	2.51	2.67	3.17	3.97
Ratio of Processors Needed by Alg 3.1 vs. BFGS, both with Analytic Gradients	2	3	4	5	6	11	21
Ratio of Processors Needed by Alg 3.1 vs. BFGS, both with Finite Diff. Gradients	1.95	2.86	3.71	4.52	5.29	8.38	11.00

Table 3.2--Average Speedup of Algorithm 3.1 over Parallel BFGS Method, n=40

q	1	2	3	4	5	10	20	40
Average Speedup	1.54	1.95	2.16	2.18	2.31	2.46	2.92	2.44
without problem 16	1.88	2.07	2.23	2.39	2.50	3.12	3.76	5.37
Ratio of Processors Needed by Alg 3.1 vs. BFGS, both with Analytic Gradients	2	3	4	5	6	11	21	41
Ratio of Processors Needed by Alg 3.1 vs. BFGS, both with Finite Diff. Gradients	1.98	2.93	3.85	4.76	5.63	9.66	15.88	21.00

Table 3.3 -- Average Speedup of Algorithm 3.1 over Parallel Newton's Method

q	1	2	3	4	5	10	20	40
Average Speedup, n=20	1.98	1.52	1.42	1.43	1.55	1.23	0.87	--
Average Speedup, n=40	2.27	2.03	1.70	1.87	1.55	1.08	0.91	0.37
without problem 16	2.70	2.23	1.71	2.02	1.63	1.37	1.33	0.94

processors. The ratios of the number of processors required by Algorithm 3.1 to the number required by BFGS are given in Tables 3.1-3.2 for both scenarios. It should be kept in mind that in the finite difference case, the BFGS method which is the baseline is already a parallel algorithm that uses the speculative gradient evaluation discussed in Section 2.2. It achieves an average speedup of 17.5 and 34.3 over the sequential, one processor BFGS algorithm in the cases $n = 20$ and $n = 40$, respectively. (This indicates that the average number of trial points evaluated per iteration in the line search is about 1.2.) Thus Algorithm 3.1 actually achieves average speedups over the sequential BFGS method of 32.6 to 69.5 for q ranging from 1 to n when $n = 20$, and 52.8 to 83.7 (64.5 to 184.2 without problem 16) for q ranging from 1 to n when $n = 40$.

There are two important conclusions from Tables 3.1-3.2. First, the new methods clearly derive a considerable gain in speed from the extra Hessian information that they use. Second, this gain is not usually proportional to the ratio of processors (or equivalently, pieces of derivative information per iteration) that they use. However, this was to be expected since we know that Newton's method, which uses roughly $n/2$ times as much information (and n or $n/2$ times as many processors) as the BFGS method, is not usually $n/2$ times as fast in terms of the number of trial points, or iterations, required. In fact on these test sets, (finite difference) Newton's method is, on the average, 4.7 and 7.1 times as fast as the BFGS method in the cases $n = 20$ and $n = 40$, respectively. The new method does a reasonable job of obtaining an increasing speedup as q changes from 1 to n. What is most satisfying is that the speedups are quite substantial for small values of q before leveling off; they are at least 50% of optimal for q up to about 4.

There is one test problem, #16 (Variably Dimensioned Problem), where the performance of Algorithm 3.1 is considerably worse than in any other case, especially when $n = 40$. This is the only problem where the performance of Algorithm 3.1 with $q=n$ is substantially worse than Newton's method, and the case $q=1$ has by far the worst performance of any test problem relative to the BFGS. We are continuing to study our algorithm to attempt to understand this behavior and see if it can be avoided. Since this one problem so strongly influences our average statistics in the case $n = 40$, Tables 3.2 and 3.3 also show what the averages would be without problem 16.

Table 3.3 compares the performance of Algorithm 3.1, with various values of q, to the performance of a parallel implementation of the finite difference Newton's method, under the assumption that the

gradient is evaluated by finite differences and that there are just enough processors to evaluate the function, finite difference gradient, and q columns of the finite difference Hessian simultaneously. This means $p = (n+1-\frac{q}{2})(q+1)$. The parallel finite difference Newton's method is assumed to use the most efficient parallel strategy. That is, at each trial point it computes the function, gradient, and as many elements of the Hessian as the remaining processors allow. Then if the trial point is accepted as the next iterate, it uses $N_q - 1$ concurrent function evaluation steps to evaluate the remainder of the Hessian, where $N_q = \left\lceil \frac{(n^2+3n+2)/2}{(n+1-(q/2))(q+1)} \right\rceil$. Thus the total number of concurrent function evaluation steps required by the parallel finite difference Newton's method to solve a particular problem is

$$(N_q \times (1 + \text{number of iterations}) + (\text{number of unsuccessful trial points}) , \qquad (3.19)$$

while for Algorithm 3.1 it is the total number of trial points for that problem. (Recall that the total number of trial points for a problem is 1 + number of iterations + number of unsuccessful trial points.)

For each value of q and n, the speedup shown in Table 3.3 is the total number of concurrent function evaluations required by Newton's method, measured by (3.19), divided by the total number of concurrent function evaluation steps required by Algorithm 3.1, where the totals are taken over all the problems successfully solved by both methods. Table 3.3 shows that for all values of $q \leq n/2$, Algorithm 3.1 is more efficient than a parallel finite difference Newton's method, under the above assumptions. (If problem 16 is included for $n = 40$ then the $q=n/2$ case is slightly worse than Newton's method on the average, but without it it is considerably better.) Thus for $q \leq n/2$, it appears to be better to evaluate just as much of the finite difference Hessian per iteration as the processors allow in one concurrent function evaluation step, rather than using extra concurrent function evaluation steps to evaluate the remainder of the Hessian.

Table 3.3 also shows that when $q = n$, Algorithm 3.1 is slightly inferior to Newton's method. The two methods are very similar in this case, since each computes the full finite difference Hessian at each iteration. The only difference is that Algorithm 3.1 does not use all this information if the approximation is indefinite, while the finite difference Newton's method uses the entire Hessian and employs the perturbed Cholesky decomposition given in Gill, Murray, and Wright [1981] to compute the search direction when the Hessian is indefinite. As might be expected, often there is no difference between the two methods but occasionally discarding some Hessian information is somewhat detrimental to the performance of the

Algorithm 3.1. It might be advantageous to incorporate a scheme for using indefinite finite difference Hessian information (perhaps the PSB or SR1 update) into Algorithm 3.1, but this would need to be done in a way that doesn't hurt the performance of the method for small q. We consider this a topic for further research.

Finally, we also tested in detail the version of Algorithm 3.1 that uses the sequential scheme (3.6), rather than the multiple update scheme (3.7), to incorporate the finite difference Hessian information. In general, the multiple update approach required from 5% to 25% fewer iterations and function evaluations to solve the same problems with the same value of q. For this reason, we recommend the multiple updating scheme.

4. Using Gradient Values at Unsuccessful Trial Points

In this section we discuss a relatively minor improvement that can be made to the parallel BFGS algorithm discussed in Section 2 as well as to some sequential BFGS algorithms. It is to use the gradient values that are computed at the unsuccessful trial points in the line search to reduce the total amount of work required to solve the optimization problem. If the gradient is evaluated by finite differences, then we assume that we are using the parallel BFGS algorithm of Section 2 with $p \geq n+1$ so that the entire finite difference gradient is evaluated at each trial point. If the analytic gradient is a by-product of the function evaluation on one processor, then the information we consider is available in a standard sequential BFGS method. It is also available in any sequential unconstrained optimization code that requires the gradient to be returned along with the function value; some unconstrained optimization software packages, for example CONMIN (Shanno and Phua [1978]) and MINOS (Murtaugh and Saunders [1983]) are organized in this way. The strategies discussed in this section are applicable to all these situations.

In all the above cases, even though the gradient is available at unsuccessful trial points, it is only used in the line search. Schnabel [1987] proposed several further uses that might be made of this information. Here we pursue the suggestion from Schnabel [1987] that we consider most promising. To facilitate our discussion, let us use the simplified notation that the current iterate is x_c, the current gradient is g_c, the current Hessian approximation is B_c, the current search direction is $d_c = -B_c^{-1} d_c$, the current trial point is

$x_t = x_c + \lambda d_c$, and the gradient at x_t is g_t. We assume that x_t is an unsatisfactory choice for the next iterate.

We will attempt to use the gradient at the unsuccessful trial point x_t to immediately update the Hessian approximation and compute a new search direction, even though we have not successfully concluded the current line search. To motivate this strategy, consider the case when f is a positive definite quadratic. It is still possible that x_t is unsatisfactory because the Hessian approximation B_c is inadequate. The standard BFGS algorithm would continue the line search until it calculates a satisfactory next iterate $x_+ = x_c + \bar{\lambda} d_c$, where $\bar{\lambda} = \sigma\lambda$ for some $\sigma \neq 1$. Let g_+ be the gradient at x_+, and let B_+ be the BFGS update to B_c using the step from x_c to x_+. Also consider the matrix B_t that would be generated as the BFGS update to B_c using the step from x_c to x_t.

The first key point is that $B_+ = B_t$, that is the updates obtained from using x_+ and g_+, or using x_t and g_t, are the same. This is because any two points along a line will generate the same secant equation, and hence the same update, for a quadratic function. (Algebraically, $x_+ - x_c = \sigma(x_t - x_c)$ and $g_+ - g_c = \sigma(g_t - g_c)$.) The other key point is that since

$$x_+ - B_+^{-1} g_+ = x_c - B_+^{-1} g_c = x_c - B_t^{-1} g_c = x_t - B_t^{-1} g_t$$

with the first and third equalities coming from the secant equation and the second from $B_+ = B_t$, we do not have to compute x_+, or adopt x_t as the new current iterate, to undertake the next iteration. Rather we can replace B_c with B_t and continue iterating from x_c, in the new direction $-B_t^{-1} g_c$. If a steplength of one is used at the new iteration, then the same point x_{++} will be generated as if we had iterated from either x_+ using the direction $-B_+^{-1} g_+$, or from x_t using the direction $-B_t^{-1} g_t$.

For quadratic f, this strategy allows a BFGS algorithm to use only one trial point per update, while likely requiring no more iterations than the standard BFGS method. If one is using the standard sequential BFGS method, Algorithm 2.1, and the gradient is being evaluated by finite differences, then the saving is small because the number of function evaluations per iteration is simply reduced from a maximum of $n+2$ to $n+1$. If, however, one is in any of the parallel or sequential scenarios mentioned at the start of this section, where the gradient is computed along with the function value at each trial point, then this strategy has the potential to cut the cost of some iterations in half (from two function-gradient pairs to one) which is a more significant savings. The strategy also has the appealing property that it never selects an unsatisfactory point x_t as an iterate; rather it incorporates gradient information from x_t that is equivalent to the

information we would have gotten at x_+, and then continues iterating from x_c which is the best point we have so far.

When f is not quadratic, B_t will not generally equal B_+, and the strategy of updating B_c to B_t and replacing the search direction from x_c, $-B_c^{-1} g_c$, with $-B_t^{-1} g_c$ may not be a good one. Ideally, replacing B_c with B_t would seem to be a good idea if B_t is closer to $\nabla^2 f(x_c)$ than B_c is in the direction d_c, i.e. if

$$\| (\nabla^2 f(x_c) - B_t) d_c \| \leq \| (\nabla^2 f(x_c) - B_c) d_c \| . \tag{4.1}$$

Since we don't know $\nabla^2 f(x_c)$, however, we try to determine whether B_t is a better approximation to $\nabla^2 f(x_c)$ than B_c is by seeing whether the quadratic model around x_c using B_t predicts $f(x_t)$ better than the quadratic model around x_c using B_c does, i.e. if

$$| f(x_c) + g_c^T d_c + d_c^T B_t d_c - f(x_t) | < | f(x_c) + g_c^T d_c + d_c^T B_c d_c - f(x_t) | . \tag{4.2}$$

(Note that it is not necessary to form B_t to check (4.2) since we know $B_t d_c = g_t - g_c$.) If (4.2) is satisfied, then it seems advantageous to calculate B_t and $d_t = -B_t^{-1} g_c$ and change the line search direction from x_c to d_t. Otherwise it seems better to continue the line search from x_c in the direction d_c in the normal fashion.

We have tested this approach on the same problem set as was used in Section 3.3 (see Appendix A1). We compared the normal BFGS algorithm, Alg. 2.1, to an algorithm that differs in that it updates the Hessian approximation and switches line search directions as described above if the line search finds an unsatisfactory point x_t which fails (3.17) and satisfies (4.2). (In addition x_t must satisfy $(g_t - g_c)^T d_c > 0$ to assure that the update will retain positive definiteness.) In this case the new strategy makes one other alteration so that a satisfactory next iterate will eventually be found : rather than starting the line search in the new direction d_t with a steplength of one, it chooses the initial steplength in the new direction so that the length of the next trial step is the same as if the line search had been continued in the old direction d_c. That is, if the next steplength in the direction d_c would have been $\bar{\lambda}$, it chooses the initial steplength λ in the direction d_t to be $\bar{\lambda} \| d_c \| / \| d_t \|$. If the next iterate again fails (3.17) but satisfies (4.2) and the positive curvature condition, then the Hessian and search direction is changed again with the steplength again being reduced by this mechanism; otherwise the line search is continued with the new line search direction. This approach is continued until a satisfactory next iterate is found. As soon as a satisfactory next iterate is found, the next line search starts with steplength one, as usual.

On our test set, we found that this strategy reduced the average number of trial point function evaluations needed to solve the problems by about 3% in the case $n = 20$, and by about 12% in the case $n = 40$. This reduction is indicative of the reduction in computational cost in any situation where the gradient is computed at each trail point, and function evaluation is the overriding cost. Recall from Section 3.3. that for these test problems, only about 20% of the total iterations have unsuccessful trial points, so that at least for $n = 40$ the observed savings are fairly satisfactory. If we use the ideal (and impossible in practice) test (4.1) instead of (4.2) to decide when to use our new strategy, the average saving rises to 15% in the case $n = 20$ but drops to 8% in the case $n = 40$. This indicates that there may be room for improvement in our results if we can find a better heuristic than (4.2) to decide when to invoke our strategy of switching line search directions.

While these savings are not dramatic, they point to a small improvement that can be made to the BFGS algorithm whenever the gradient is available at each trial point, i.e. both in the parallel BFGS method discussed in Section 2 and in some sequential BFGS codes. It also leads us to be interested in some related ideas that we mention in the next section.

5. Summary and Directions for Future Research

In Section 2 we have shown that it is fairly easy to efficiently utilize up to $n+1$ processors in the standard BFGS algorithm for unconstrained optimization, in two different situations. First, if function evaluation is expensive and gradients are evaluated by finite differences, then by evaluating the gradient along with the function at every trial point one can generally realize at least 70-80% efficiencies with up to $n+1$ processors. Secondly, if n is large enough that the linear algebra costs of the method are significant, then it is also fairly easy to parallelize the linear algebra efficiently for up to n processors. This causes us to reexamine the various implementations of the BFGS update, and to chose the unfactored update of the inverse matrix which appears to be the cheapest sequential and parallel approach and to have no noticeable finite precision difficulties in comparison to the other possible approaches.

Several variations on the approaches of Section 2 merit investigation. One is whether it would be better to use extra processors to evaluate multiple points simultaneously during the line search, as proposed

by several authors including Dixon [1981], Dixon and Patel [1982], Patel [1982], Lootsma [1984], and van Laarhoven [1985], rather than performing speculative finite difference gradient evaluations. A second issue is whether the modification of the BFGS method recently proposed by Powell [1987] has a significant advantage in practice, and if so, how it (different) linear algebra is best implemented on a parallel computer.

In Section 3 we have examined the situation when function evaluation is expensive and there are more processors than are needed to evaluate the function and gradient simultaneously. This occurs if the gradient is evaluated by finite differences and the number of processors, p, is greater than $n+1$, or if the gradient is evaluated analytically along with the function on one processor and $p>1$. If there are enough processors so that we can evaluate the function, gradient, and (finite difference) Hessian concurrently, then we would use a parallel implementation of Newton's method. If not, then we have proposed using new optimization algorithms that use the function, gradient, and $q<n$ columns of the finite difference Hessian at each iteration. These can be thought of as falling in between the BFGS method and Newton's method. We have shown that the performance of these methods for different values of q varies between the performance of the BFGS method and Newton's method as might be expected. We have also shown that if there are just enough processors to evaluate the function, gradient, and q columns of the finite difference Hessian, and our new method is more efficient than either the parallel BFGS method or a parallel implementation of the finite difference Newton's method.

There are several interesting research questions regarding these new methods that use part of the Hessian at each iteration. One is whether it would be better to use an update that allows indefiniteness, such as the SR1, to incorporate the finite difference information, rather than using the BFGS as we have done. This question is especially intriguing in light of the recent results of Conn, Gould, and Toint [1986] that report very good computational performance for a trust region method using the SR1. A second issue is whether some different procedure for choosing the finite difference directions $\{u_i\}$ that are used at each iteration would be preferable. An example would be choosing these directions based upon the recent step directions. Another more general issue is whether there are better ways to utilize additional processors than evaluating part of the Hessian, for example using the extra evaluations to form a higher order model such as the tensor model introduced by Schnabel and Frank [1984].

In Section 4 we have considered the use of gradient information at unsuccessful trial points in the line search. Such information is available in our parallel methods that evaluate function and gradient information simultaneously, and also in several well-known sequential unconstrained optimization packages. We have shown that we can modify the BFGS algorithm to utilize this information in a way that leads to small gains in efficiency.

This work on using derivative information from unsuccessful trial points might be extended in a number of directions. In the parallel methods that use partial Hessian information, one could also consider whether the Hessian information from unsuccessful trial points could be utilized. In this connection one might want to consider evaluating the partial Hessian information at the current iterate x_c rather than at the trial point x_t. We also have not yet considered the case when $p \leq n$ where only part, rather than all, of the finite difference gradient is evaluated at the unsuccessful trial point.

6. References

K. W. Brodlie, A. R. Gourlay, and J. Greenstadt [1973], "Rank-one and rank-two corrections to positive definite matrices expressed in product form," *Journal of the Institute of Mathematics and its Applications* 11, pp. 73-82.

R. H. Byrd and J. Nocedal [1987], "A tool for the analysis of quasi-Newton methods with application to unconstrained minimization," Technical Report ANL/MCS-TM-103, Mathematics and Computer Science Division, Argonne National Laboratory.

R. H. Byrd, R. B. Schnabel, and G. A. Shultz [1987], "Using parallel function evaluations to improve Hessian approximations for unconstrained optimization," Technical Report CU-CS-361-87, Department of Computer Science, University of Colorado at Boulder.

A. R. Conn, N. I. M. Gould, and Ph. L. Toint [1986], "Testing a class of methods for solving minimization problems with simple bounds on the variables," Research Report CS-86-45, Faculty of Mathematics, University of Waterloo, Waterloo, Canada.

W. C. Davidon [1975], "Optimally conditioned optimization algorithms without line searches", *Mathematical Programming* 9, pp. 1-30.

J. E. Dennis Jr. and J. J. Moré [1974], "A characterization of superlinear convergence and its application to quasi-Newton methods", *Mathematics of Computation* 28, pp. 549-560.

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

J. E. Dennis Jr. and R. B. Schnabel [1987], "A view of unconstrained optimization," Technical Report CU-CS-376-87, Department of Computer Science, University of Colorado at Boulder, to appear in *Handbooks in Operations Research and Management Science, Vol. 1, Optimization,* G. L. Nemhauser, A. H. J. Rinnooy Kan, and M. J. Todd, eds., North-Holland, Amsterdam.

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

L. C. W. Dixon and K. D. Patel [1982], "The place of parallel computation in numerical optimisation IV, parallel algorithms for nonlinear optimisation", Technical Report No. 125, 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, G. H. Golub, W. Murray, and M. A. Saunders [1974], "Methods for modifying matrix factorizations," *Mathematics of Computation* 28, pp. 505-535.

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

D. Goldfarb [1976], "Factorized variable metric methods for unconstrained optimization", *Mathematics of Computation* 30, pp. 796-811.

L. Grandinetti [1978], "Factorization versus nonfactorization in quasi-Newtonian methods for differentiable optimization," Report N5, Dipartimento di Sistemi, Universita della Calabria.

S. P. Han [1986], "Optimization by updated conjugate subspaces," in *Numerical Analysis: Pitman Research Notes in Mathematics Series 140*, D.F. Griffiths and G.A. Watson, eds., Longman Scientific and Technical, Burnt Mill, England, pp. 82-97.

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

J. J. Moré, B. S. Garbow, and K. E. Hillstrom [1981], "Testing unconstrained optimization software", *ACM Transactions on Mathematical Software* 7, pp. 17-41.

J. J. Moré and D. C. Sorensen [1983], "Computing a trust region step", *SIAM Journal on Scientific and Statistical Computing* 4, pp. 553-572.

B. A. Murtagh and M. A. Saunders [1983], "MINOS 5.0 User's Guide," Technical Report SOL 83-20, Department of Operations Research, Stanford University.

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.

M. J. D. Powell [1976], "Some global convergence properties of a variable metric method without exact line searches", in *Nonlinear Programming*, R. Cottle and C. Lemke, eds. AMS, Providence, R.I., pp. 53-72.

M. J. D. Powell [1987], "Updating conjugate directions by the BFGS formula," *Mathematical Programming* 38, pp. 29-46.

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 [1987], "Concurrent function evaluations in local and global optimization," *Computer Methods in Applied Mechanics and Engineering* 64, pp. 537-552.

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

R. B. Schnabel, J. E. Koontz, and B. E. Weiss [1985], "A modular system of algorithms of unconstrained minimization", *ACM Transactions on Mathematical Software* 11, pp. 419-440.

D. F. Shanno and K. H. Phua [1978a], "Matrix conditioning and nonlinear optimization," *Mathematical Programming* 14, pp. 145-160.

P. J. M. van Laarhoven [1985], "Parallel variable metric methods for unconstrained optimization," *Mathematical Programming* 33, pp. 68-81.

Table A1 -- Test Problem Set

Problem Number	Problem Name	Source of Problem
1	Trigonometric	MGH26
2	Extended Rosenbrock	MGH21
3	Extended Powell Singular	MGH22
4	Chebyquad	MGH35
5	Chained Singular	CGT5
6	Generalized Wood	CGT7
7	Chained Wood	CGT8
8	Generalized Broyden Tridiagonal (a)	CGT10
9	Generalized Broyden Tridiagonal (b)	CGT11
10	Generalized Broyden Banded (a)	CGT12
11	Generalized Broyden Banded (b)	CGT13
12	Toint-Broyden 7 Diagonal	CGT14
13	Toint Trigonometric	CGT16
14	Generalized Cragg and Levy	CGT17
15	Generalized Brown	CGT21
16	Variably Dimensioned	MGH25
17	Penalty Function I	MGH23
18	Penalty Function II	MGH24

CGT = Conn, Gould, Toint [1986]
MGH = Moré, Garbow, Hillstrom [1981]

Table A2 -- Test Results, n=20

Problem Number	Iterations Unsuccessful Trial Points								
	BFGS	Algorithm 3.1							Newton's
		q=1	q=2	q=3	q=4	q=5	q=10	q=20	Method
1	47	28	27	23	23	16	12	12	12
	6	5	9	13	15	12	4	6	6
2	46	94	80	84	72	68	45	24	24
	21	19	26	38	55	44	16	8	8
3	48	75	47	41	36	26	24	15	15
	19	1	2	4	6	0	4	0	0
4	54	49	47	--	32	34	--	--	--
	15	8	13	--	12	19	--	--	--
5	308	57	41	37	31	29	27	20	20
	21	4	2	3	1	0	0	0	0
6	164	133	134	103	107	101	86	54	51
	32	23	56	48	49	77	50	38	28
7	271	50	118	66	58	51	69	48	49
	33	10	45	37	21	14	59	27	27
8	56	34	24	20	16	16	14	10	10
	4	0	0	0	0	0	0	0	0
9	21	20	14	13	11	10	7	5	5
	3	0	0	0	0	0	0	0	0
10	125	32	25	20	18	17	15	12	12
	6	0	0	0	0	1	0	0	0
11	107	22	15	13	12	10	9	7	7
	5	0	0	0	0	0	0	0	0
12	58	27	18	15	12	12	9	6	6
	5	0	0	0	0	0	0	0	0
13	42	36	18	19	25	19	18	11	8
	113	98	19	17	32	25	17	13	3
14	141	43	--	--	--	--	--	--	19
	13	14	--	--	--	--	--	--	0
15	6	8	8	8	7	6	5	4	4
	1	0	0	0	0	0	0	0	0
16	21	117	72	38	42	39	30	50	18
	7	4	2	3	0	5	8	15	0
17	140	91	72	51	56	51	45	33	33
	52	12	1	4	1	2	0	5	4
18	226	89	90	72	77	75	58	60	62
	42	13	14	6	11	5	7	23	25

-- = overflow

Table A3 -- Test Results, n=40

Problem Number	Iterations Unsuccessful Trial Points									
	BFGS	Algorithm 3.1								Newton's Method
		q=1	q=2	q=3	q=4	q=5	q=10	q=20	q=40	
1	84	46	68	62	53	48	17	17	15	--
	1	12	22	36	26	13	4	18	29	--
2	47	129	102	108	90	97	83	46	24	24
	22	36	39	84	77	69	70	30	8	8
3	48	95	70	56	49	45	25	24	15	15
	19	0	1	3	9	37	0	2	0	0
4	--	248	240	130	180	164	--	--	--	--
	--	34	36	31	51	40	--	--	--	--
5	300	95	56	50	39	33	32	28	20	20
	22	7	0	0	0	1	5	5	0	0
6	194	217	203	155	152	140	108	122	58	--
	32	55	69	99	147	120	112	86	35	--
7	**	97	146	152	42	36	77	77	51	49
	**	23	97	93	14	11	45	53	36	32
8	59	48	35	28	24	22	19	14	10	10
	4	0	0	0	0	0	5	0	0	0
9	41	27	21	17	15	13	10	7	5	5
	22	0	0	0	0	0	0	0	0	0
10	172	47	33	28	25	22	18	15	12	12
	8	0	0	0	0	0	0	0	0	0
11	74	27	24	18	15	13	11	9	7	7
	5	0	0	0	0	0	0	0	0	0
12	116	36	26	22	19	17	12	9	6	6
	6	0	0	0	0	1	0	0	0	0
13	68	23	22	20	24	26	26	25	24	--
	229	22	19	14	23	24	38	39	58	--
14	149	61	--	--	34	34	--	--	22	19
	14	12	--	--	27	30	--	--	11	0
15	6	8	8	9	9	9	6	5	4	4
	1	0	0	0	0	0	0	0	0	0
16	27	292	170	74	101	86	130	120	259	22
	7	6	1	11	9	6	74	60	287	0
17	142	144	90	70	64	69	55	46	34	36
	41	3	4	6	1	5	2	0	7	7
18	419	87	49	46	45	38	33	27	23	23
	21	14	7	10	13	10	5	1	1	1

** = iteration limit (500), -- = overflow

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CU-CS-396-88	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION University of Colorado 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION U.S. Army Research Office
6c. ADDRESS (City, State and ZIP Code) Computer Science Department Campus Box 430 Boulder, CO 80309-0430	7b. ADDRESS (City, State and ZIP Code) Post Office Box 12211 Research Triangle Park, NC 27709
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER DAAG-29-84-K-0140
8c. ADDRESS (City, State and ZIP Code)	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT NO.
11. TITLE (Include Security Classification) Parallel Quasi-Newton Methods for Unconstrained Optimization	
12. PERSONAL AUTHOR(S) Richard H. Byrd, Robert B. Schnabel, Gerald A. Shultz	
13a. TYPE OF REPORT Technical	13b. TIME COVERED FROM _____ TO _____
14. DATE OF REPORT (Yr., Mo., Day) 88/04/01	15. PAGE COUNT 41
16. SUPPLEMENTARY NOTATION	
17. COSATI CODES FIELD / GROUP / SUB. GR.	18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) Parallel computation, unconstrained optimization, quasi-Newton methods

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

We discuss methods for solving the unconstrained optimization problem on parallel computers, when the number of variables is sufficiently small that quasi-Newton methods can be used. We concentrate mainly, but not exclusively, on problems where function evaluation is expensive. First we discuss ways to parallelize both the function evaluation costs and the linear algebra calculations in the standard sequential secant method, the BFGS method. Then we discuss new methods that are appropriate when there are enough processors to evaluate the function, gradient, and part but not all of the Hessian at each iteration. We develop new algorithms that utilize this information and analyze their convergence properties. We present computational experiments showing that they are superior to parallelization of either the BFGS method or Newton's method under our assumptions on the number of processors and cost of function evaluation. Finally we discuss ways to effectively utilize the gradient values at unsuccessful trial points that are available in our parallel methods and also in some sequential software packages.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ OTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Dr. Jagdish Chandra	22b. TELEPHONE NUMBER (Include Area Code) 619/549-0641 22c. OFFICE SYMBOL

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.