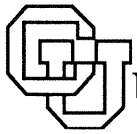
PARALLEL NONLINEAR OPTIMIZATION: LIMITATIONS, OPPORTUNITIES, AND CHALLENGES

Robert B. Schnabel

CU-CS-715-94



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Parallel Nonlinear Optimization: Limitations, Opportunities, and Challenges

Robert B. Schnabel

CU-CS-715-94 March 1994

Department of Computer Science
Campus Box 430
University of Colorado,
Boulder, Colorado, 80309-0430 USA
email: bobby@cs.colorado.edu

This research was supported by AFOSR Grant No. AFOSR-90-0109, ARO Grant No. DAAL03-91-G-0151, NSF Grant No. CCR-9101795, and NSF Grant No. CDA-8922510.

Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the agencies named in the acknowledgements section.

Parallel Nonlinear Optimization: Limitations, Opportunities, and Challenges

Abstract

The availability and power of parallel computers is having a significant impact on how large-scale problems are solved in all areas of numerical computation, and is likely to have an even larger impact in the future. This paper attempts to give some indication of how the consideration of parallel computation is affecting, and is likely to affect, the field of nonlinear optimization. It does not attempt to survey the research that has been done in parallel nonlinear optimization. Rather it presents a set of examples, mainly from our own research, that is intended to illustrate many of the limitations, opportunities, and challenges inherent in incorporating parallelism into the field of nonlinear optimization. These examples include parallel methods for small to medium size unconstrained optimization problems, parallel methods for large block bordered systems of nonlinear equations, and parallel methods for both small and large-scale global optimization problems. Our overall conclusions are mixed. For generic, small to medium size problems, the consideration of parallelism does not appear to be leading to major algorithmic innovations. For many classes of large-scale problems, however, the consideration of parallelism appears to be creating opportunities for the development of interesting new methods that may be advantageous for parallel and possibly even sequential computation. In addition, a number of large-scale parallel optimization algorithms exhibit irregular, coarse-grain structure, which leads to interesting computer science challenges in their implementation.

1. Introduction

Parallel computation is having a significant impact upon how large scale scientific computation is performed. To solve the large scientific problems of interest to scientists and engineers today, very powerful computers are necessary, and it appears that many if not all of the most powerful scientific computers of the future (as well as at present) will be parallel computers. To scientific computation researchers, one of the most interesting aspects of this transition from sequential or vector computers to parallel computers is that new algorithm development may be required to use parallel machines efficiently.

In this paper, we attempt to give some indication of how the move to parallel computers is affecting, and is likely to affect, the field of nonlinear optimization. Our focus is on whether, and where, the use of parallel computers is leading to interesting new algorithmic approaches for nonlinear optimization. To address this issue, we try to point out some of the key limitations, opportunities, and challenges in developing parallel algorithms for nonlinear optimization problems. Our overall conclusion is mixed: in some portions for nonlinear optimization, primarily generic methods for small to medium scale problems, the consideration of parallelism does not appear to be leading to major algorithmic changes or challenges, while in other areas, primarily classes of large-scale optimization problems, there appear to be very interesting opportunities and challenges that arise from the consideration of parallelism.

On the parallel architecture side, our focus in this paper is on moderately to massively parallel MIMD computers. This broad class of machines appears to be the main architectural direction that the parallel computation field is pursuing for general purpose, high performance computation. Section 2 gives a brief overview of parallel computation that should be sufficient for the purposes of this paper. We briefly summarize current trends in parallel computer architectures, and the relationship between parallel computers and the needs of nonlinear optimization methods. This section is by no means intended to serve as a tutorial in parallel computation; some references that fulfill this role include [1,18,29].

In nonlinear optimization, our focus in this paper is on continuous, nonlinear problems. These include solving unconstrained and constrained optimization problems, and systems of nonlinear equations. (We assume that the reader of this paper is familiar with the basics of nonlinear optimization as found in [11,14,17].) It is not, however, our intention in this paper to present a survey of parallel algorithm research in these areas. Rather we will present a small number of examples, primarily from our own research, that illustrate the limitations, opportunities, and challenges in parallel nonlinear optimization. Section 3 discusses parallel methods for unconstrained optimization problems. It primarily illustrates the limitations in developing novel generic parallel optimization algorithms for moderate size problems. Section 4 discusses parallel methods for a special class of large scale problems, block-bordered systems of nonlinear equations. This problem class illustrates the possibility for the consideration of parallelism to lead to the development of new methods that are advantageous on sequential as well as parallel computers. Then in Section 5 we describe parallel methods for large global optimization problems. Section 5 illustrates that large-scale parallel optimization algorithms can be particularly interesting and challenging because they are often irregular, coarse-grain algorithms that are challenging both to develop and to implement. Finally, Section 6 briefly summarizes our views of the limitations, opportunities, and challenges in parallel nonlinear optimization.

This paper is based upon lectures given at the NATO Advanced Research Institute on Algorithms for Continuous Optimization at Il Ciocco, Italy in September 1993. I thank the organizers and participants of the institute for the stimulating environment that it provided. The parallel optimization research described in Sections 3-5 that was performed at the University of Colorado is all joint work; the

participants in various projects include Richard Byrd, Tom Derby, Cees Dert, Betty Eskow, Humaid Khalfan, Bart Oldenkamp, Jerry Shultz, Andre van der Hoek, Alexander Rinnooy Kan, Chung-Shang Shao, Sharon Smith, and Xiaodong Zhang.

2. Parallel Computation Background, with Relations to Parallel Optimization

The primary motivation for parallel computation is the continuing need for faster and larger computers, in terms of computational speed, memory and input/output capacity. While computer speeds and memory sizes have continued to increase dramatically, many important scientific problems still require orders of magnitude greater speed and/or memory than is currently available. For decades, these increases in speed and memory were attained by building ever faster and larger sequential electronic computers. However, the evolution of these computers has now reached the stage where further improvements are constrained by fundamental physical limits (e.g. the speed of light, the sizes of atoms) and the limitations these place upon basic machine characteristics such as minimal feature sizes, minimal distances between components, and maximal rates of heat dissipation. Also, very powerful processor/memory chips recently have become widely and cheaply available. These two trends, the limits of sequential computation speed and the availability of commodity chips, along with the continuing need for vastly increased computing power, are largely responsible for the greatly increased production of parallel computers in the last five years.

The challenges in developing and effectively utilizing parallel computers come from many sides: hardware, systems and language software, and applications algorithms and software. The discussion of parallel applications algorithms for optimization is the main focus of this paper, in Sections 3-5, and a few comments on parallel systems and language support are included in these sections. This section briefly discusses parallel architectures, concentrating on those that appear to be of greatest interest to nonlinear optimization. To motivate this, we first briefly consider the motivation and needs for parallel optimization.

2.1 Possibilities for Parallelism in Optimization

Parallelism is of interest in optimization because many optimization problems are expensive to solve. To determine what sort of parallel computers may be of interest, it is useful to examine where the expense in nonlinear optimization algorithms comes from. There are at least four different possible sources:

1. The nonlinear functions, constraints, and/or derivatives may be expensive to evaluate.
2. The number of variables or constraints, and hence the cost of each iteration aside from function and derivative evaluation, may be large.
3. Many evaluations of the objective function, constraints, or derivatives may be required.
4. Many iterations may be required.

These in turn lead to at least three levels at which one may consider introducing parallelism into an optimization algorithm:

1. Parallelize each evaluation of the objective function, constraints, and/or their derivatives.
2. Parallelize the linear algebra involved in each iteration.
3. Parallelize the optimization process at a high level, either to perform multiple function, constraint, and/or derivative evaluations on multiple processors concurrently, and/or to reduce the total number of iterations required.

All of these are important ways to create parallel optimization methods, and any may be the best way to utilize parallel computation for a given problem. In this paper, however, we are mainly interested in the third possibility, parallelizing the optimization process, since this is the domain of optimization researchers. The second possibility, parallelizing the linear algebra, may be the concern of optimization researchers if the linear algebra is particular to optimization algorithms. The first possibility is likely to be outside the domain of optimization research; for example if the objective function evaluation involves the solution of a system of differential equations, this option would entail creating or using a parallel differential equations solver.

As will be seen in Sections 3-5, parallelizing the optimization process in a nonlinear optimization algorithm is likely to lead to a "coarse-grain" parallel algorithm. By this we mean an algorithm where each processor performs a significant amount of computation in between each point where it communicates or synchronizes with other processors. For example, if each processor performs at least one function evaluation between communication points, and these function evaluations are even moderately expensive, a coarse-grain parallel algorithm results. Such parallel algorithms are generally well suited to MIMD computers, and less well suited to SIMD computers or vector computers. This is one reason why our brief survey of parallel architectures in Section 2.2 concentrates on MIMD computers.

2.2 Brief Survey of Parallel Architectures

The main classes of parallel computers that are currently used for scientific computation are shared and distributed memory MIMD multiprocessors, and SIMD processor arrays. We briefly discuss the key characteristics of these types of computers in this section, and their main advantages and limitations. We omit discussion of some other architectures, including data-flow computers and systolic arrays, that are not currently in wide use for scientific computation.

An MIMD (Multiple Instruction Multiple Data) computer is a computer with multiple processors that can execute different instructions on different data at the same time. This means that such a computer can execute multiple, similar or dissimilar tasks concurrently. All MIMD computers include multiple processors (arithmetic and instruction processing units) and some method of communicating between them. In a shared memory multiprocessor, the processors all share access to a global memory, and communicate by reading and writing data residing in this memory. Generally, the shared memory is large, many million bytes. In addition, the processors generally each have much smaller (typically 10,000-100,000 byte) local memories. In a distributed memory multiprocessor, there is no shared memory. Instead, each processor has a large local memory (typically 1-100 million bytes), and the processors communicate by sending messages to each other over a network that connects them.

Shared memory multiprocessors have two fundamental advantages over distributed memory multiprocessors. First, communication via the shared memory is usually considerably faster than communication by messages in a distributed memory multiprocessor. This enables a wider range of parallel algorithms, in particular those with higher communication/computation ratios, to run efficiently. Secondly, shared memory machines generally are easier to program and utilize, since the programmer needs to give less attention to communication and data partitioning. The main limitation of shared memory multiprocessors is that they appear difficult to build with large numbers of processors. The difficulty is the cost and complexity of providing all the processors with uniform, fast access to the global shared memory. Currently, shared memory multiprocessors are being built with two to about 32 processors by many vendors including Cray, but only distributed memory architectures are providing computers with hundreds or thousands of processors. (An in-between class, virtual shared memory multiprocessors, is addressed

shortly.) One rather different and interesting alternative for providing a shared memory multiprocessor that may scale effectively is the pipelined instruction and memory paradigm that is embodied in the Tera computer currently under development; the description of this architecture is beyond the scope of this paper.

Among distributed memory multiprocessors, one can currently identify at least three subclasses: "pure" distributed memory multiprocessors, networks of computers, and virtual shared memory multiprocessors. In a pure distributed memory multiprocessor, each processor's memory has its own address space, and all communication is by message passing (at least at the hardware level). The main advantages and disadvantages of pure distributed memory multiprocessors are exactly the reverse of those for shared memory multiprocessors. The main advantage is that pure distributed memory multiprocessors are relatively easy to construct and to scale to hundreds or thousands of processors; machines of this size have been offered by several vendors including IBM, Intel, Meiko, NCUBE, and Thinking Machines. Typically each processor is powerful, with roughly the capability of a modern scientific workstation. The interconnection network is generally a hypercube or a two-dimensional grid; this will not be of importance in this paper. The first main disadvantage of pure distributed memory multiprocessors is that communication is generally quite slow, with a typical ratio of the time to communicate a number to the time to do one floating point operation being one thousand. This means that parallel algorithms must have fairly coarse granularity to run efficiently on these machines. A second key disadvantage is that these machines are rather difficult to program, due to the need to explicitly partition data structures among the processors and manage communication. Current research in parallel programming languages, such as the High Performance Fortran and Fortran D projects ([15,36]), is attempting to diminish or remove the latter disadvantage.

Networks of computers can function as distributed memory multiprocessors. All that is needed is some mechanism for communicating between multiple machines on the network, a facility that is provided by most modern operating systems. In comparison to pure distributed memory multiprocessors, using a networks of computers as a multiprocessor has the advantages that it can utilize existing computational resources, and that the ratio of hardware cost to floating point speed is lower. The primary disadvantage is that communication is even slower than on a pure distributed memory multiprocessor. Generally this means that parallel algorithms must have very coarse granularity (thousands of computations between communication points) to be efficient in this environment.

Virtual shared memory multiprocessors are a recently introduced class of computers that may be considered to reside between shared and pure distributed memory multiprocessors. Physically, they are distributed memory machines: each processor has its own memory and there is no global shared memory. However, there is a single address space for all the memories, and any processor may directly access any memory location in any processor's memory. If the access is not to the local memory, the data is retrieved by the hardware. As expected, the advantages of this architecture lie between those of shared and pure distributed memory machines. Communication costs are expected to be in between the two extremes, and the programmer may not have to manage the placement of data, although there may be advantages to doing so. A key question is the scalability of this class of architectures, i.e. how many processors can be supported effectively. Kendall Square Research has produced such a machine with up to 128 processors; it remains to be seen whether efficient virtual shared memory machines can be built with considerably higher numbers of processors.

Finally, a SIMD (Single Instruction Multiple Data) computer is a computer with multiple processors that all can execute the same instruction on different data at the same time. This means that such a

computer can execute a given program segment simultaneously (in lockstep) on multiple processors using multiple data sets, so long as the program segment contains no data dependent branches. One addition to this model that is generally supported is that some processors may do nothing ("mask") rather than execute a given instruction. Architecturally, such computers consist of a control processor and its memory, which generates the sequence of instructions that all processors follow; multiple processors, each with a local memory, that execute these instructions on their data; and a network that connects all the memories and allows communication between them. A number of such computers have been built by several vendors including MassPar and Thinking Machines. Their advantages are that they are relatively easy to build with large number of processors, and to program. Their primary disadvantage is that the SIMD programming model has limited applicability. For example, it would not usually be possible to evaluate a nonlinear function $f(x)$ at a different point x simultaneously on each processor of such a computer, because most nonlinear function evaluation codes include data dependent branches.

2.3 Summary and Relation to Optimization

Among the parallel computers available today, the ones that appear suitable for general purpose optimization algorithms are the various types of MIMD machines: shared memory multiprocessors, virtual shared memory multiprocessors, distributed memory multiprocessors, and networks of computers used as multiprocessors. In considering these four classes of machines, it is important to be aware of several differences between them. First, the ratio of communication speed to (floating point) computational speed generally increases monotonically in the order that the classes are listed above. Second, the ease of producing multiprocessors of these types arguably also increases monotonically in the order they are listed. Third, the ease of programming these machines arguably decreases monotonically in the order they are listed above.

Because many parallel optimization algorithms have very coarse granularity, they are often well suited to any type of MIMD computer. On the other hand, the SIMD architecture is not general enough for most parallel optimization algorithms, particularly those involving multiple, concurrent evaluations of an arbitrary nonlinear objective function, although it may be well suited to parallelizing the linear algebraic calculations in optimization algorithms. For these reasons, the remainder of this paper is oriented to parallel computation on MIMD multiprocessors, usually without being specific about the type of MIMD multiprocessor.

3. General Purpose Parallel Methods for Small to Medium Size Problems -- Unconstrained Optimization

This section illustrates some of the opportunities and limitations that arise in creating general purpose parallel algorithms for small to medium size optimization problems. It does so by considering the example of the unconstrained optimization problem

$$\underset{x \in R^n}{\text{minimize}} \; f(x) : R^n \rightarrow R \quad .$$

The most commonly used approach for solving unconstrained optimization problems when the number of variables is not too large, say less than 100, is probably the BFGS quasi-Newton method. In this section we will discuss the opportunities and limitations in creating parallel quasi-Newton methods. This section draws extensively on material in [7].

Like all methods for finding local minimizers, quasi-Newton methods are iterative. Given a current iterate x_c, the function and gradient values at this iterate, and a current Hessian approximation A_c, the basic steps of a quasi-Newton method are:

1. Calculate the search direction d_c: Solve $A_c \cdot d_c = -\nabla f(x_c)$ for d_c.
2. Line search: Find a $\lambda>0$ for which $f(x_c + \lambda d_c) < f(x_c)$.
3. Calculate $\nabla f(x_c + \lambda d_c)$ and decide whether to stop; if not
4. Update the Hessian approximation: $A_+ = A_c +$ rank-two matrix

The BFGS method corresponds to a specific choice of the rank-two matrix in step 4, but this formula is not important to the discussion in this section. Also, there are various implementations of steps 1 and 4 that are mathematically equivalent but involve different methods of storing and updating A_c and solving for d_c. These variants have interesting consequences for parallelism that are discussed later in this section.

The main opportunities for using parallelism in existing or new quasi-Newton methods correspond to the three general uses of parallelism in optimization algorithms that were mentioned in Section 2.1:

1. One can parallelize the individual evaluations of $f(x)$ or $\nabla f(x)$ in steps 2 and 3 above.
2. One can parallelize the linear algebraic calculations in steps 1 and 4 above.
3. One can perform multiple evaluations of $f(x)$ (or $\nabla f(x)$) concurrently, either within the algorithmic framework above, or by devising new algorithms.

The remainder of this section discusses each of these possibilities briefly, and then draws some overall conclusions.

3.1 Parallelizing Function or Derivative Evaluations

As mentioned in Section 2.1, parallelizing the individual evaluations of $f(x)$ may be a very effective way to utilize parallel computers in optimization algorithms if the function evaluations are expensive and can be parallelized effectively, but this endeavor is outside the domain of optimization algorithm research. Parallelizing the gradient evaluations is a little more interesting to discuss since there are several possibilities. If the gradient is evaluated analytically then the same comments apply: this may be an effective use of parallelism but is outside the optimization domain. If the gradient is evaluated by finite differences, e.g. by the formula

$$\nabla f(x_c)_i \approx \frac{f(x_c + h_i e_i) - f(x_c)}{h_i} \quad , \quad i = 1, \cdots, n \tag{3.1}$$

where e_i is the i-th unit vector, then a trivial use of parallelism is to perform the n additional, independent function evaluations $f(x_c + h_i e_i)$ concurrently. If n is considerably larger than the number of processors and function evaluation is expensive, this may be all that is necessary to effectively parallelize a quasi-Newton method that uses finite difference gradients. Finally, if the gradient is evaluated by recently developed automatic differentiation techniques ([21]), there are several possibilities. If the forward version of automatic differentiation is used, analogous possibilities exists as for finite difference gradients. If the reverse mode is used, utilizing parallelism appears to be more difficult and is a topic of current research. In summary, parallelizing individual function or gradient evaluations may be an effective way to utilize parallelism for quasi-Newton methods, but it does not involve any change in the optimization algorithm.

3.2 Parallelizing Linear Algebraic Calculations

Next we turn to the issue of utilizing parallelism during the linear algebraic calculations of a quasi-Newton method. This may be desirable if the number of variables is reasonably large. While this topic also may appear to be outside the domain of optimization research, it is not entirely, because there are several ways of performing the algebraic calculations in steps 1 and 4 that are mathematically equivalent but have interesting differences and tradeoffs with respect to their operation counts, numerical properties, and parallelizability. For example, when the BFGS update (or any other rank-two update that preserves symmetry and positive definiteness) is used, the four main possibilities are:

1. Update A_c to A_+ by adding a rank-two matrix to A_c; each calculation of d_c requires a Cholesky factorization of A_c and two triangular solves.
2. Update a Cholesky factorization of A_c to a Cholesky factorization of A_+ by adding a rank-one matrix to the Cholesky factor of A_c and then reducing this updated matrix back to lower triangular form by a sequence of $2n-2$ Givens' rotations; each calculation of d_c then requires two triangular solves.
3. Update $(A_c)^{-1}$ to $(A_+)^{-1}$ by adding a rank-two matrix to $(A_c)^{-1}$; each calculation of d_c requires a matrix-vector multiplication.
4. Update the (non-triangular) factorization $B_c B_c^T$ of $(A_c)^{-1}$ to a (non-triangular) factorization $B_+ B_+^T$ of $(A_+)^{-1}$ by adding a rank-one matrix to B_c to obtain B_+; each calculation of d_c then requires two matrix-vector multiplications.

Options #1 and #2 maintain approximations of the Hessian, while options #3 and #4 maintain the inverses of these approximations. Options #2 and #4 keep the approximations in factored form, while options #1 and #3 keep them in unfactored form. For a detailed review of these possibilities, see [11] or [7]. Here we just briefly review their comparison and then discuss the interesting issue from the viewpoint of this paper, which is the impact that the consideration of parallelism has had upon the choice between these options. This topic is also covered in more detail in [7].

Of the four options given above, the most straightforward, #1, requires $O(n^3)$ operations, while the remainder require $O(n^2)$ operations. Thus #1 is not generally used in practice. Of the remainder, #3 is the cheapest, but #2 has been used in most production codes, because it implicitly retains positive definiteness of the Hessian approximation by maintaining a Cholesky factorization of it. This in turn guarantees that all the search directions d_c are descent directions, which is very important to optimization algorithms. It had long been feared that the matrices generated by option #3 might lose positive definiteness due to finite precision error, and then possibly generate non-descent directions, and for this reason it has not often been used in codes.

However the consideration of parallelism introduces a conflicting factor into the comparison between options #2 and #3, because option #3 is much more conducive to parallelism than option #2. Option #3 requires only matrix-vector operations (a rank-two update and a matrix-vector multiplication), which can be parallelized very nicely. On the other hand, option #2 is based upon Givens' rotations and triangular solves, which require sequences of vector-vector operations on vectors ranging from length 2 to n and parallelize very poorly.

These reasons have motivated several researchers to examine whether there really is a difference in the numerical performance of BFGS methods based on option #2 versus option #3. Tests by [7,19,27] have found negligible differences in the iterates produced, over broad sets of problems. Given the

considerable advantage of option #3 with respect to parallel implementation, it therefore appears that option #3 is preferable to option #2, at least on parallel computers. This is an interesting example where the consideration of parallelism has led to the re-examination of a part of a basic optimization algorithm, with interesting conclusions.

Finally, it should be noted that option #4 has rarely been considered, but is closely related to a method proposed by Han [20] for use in parallel computation. Option #4 has some attractive properties: like option #3, it parallelizes very well, and like option #2, it implicitly maintains positive definiteness by keeping a factorization of the matrix. However, it is more expensive than option #3. Thus, as long as option #3 does not have numerical problems, which appears to be the case, it would seem to be the preferable option for parallel (and probably also sequential) computation. Otherwise, option #4 would seem to be an attractive choice for parallel computation.

In summary, the consideration of parallelism has led to an interesting re-examination of the implementation of the linear algebraic steps in quasi-Newton methods. Note, however, that these options for parallelism do not involve any change in the basic optimization method, only in the details of its implementation.

3.3 Performing Multiple Function or Derivative Evaluations Concurrently

Last we turn to the third possibility for parallelizing a quasi-Newton method, the utilization of multiple, concurrent function or derivative evaluations. We only will consider the case when each function evaluation is performed by one processor, and the gradient is evaluated by finite differences (equation 3.1), but the points that this discussion makes are more general.

To motivate this discussion, consider the overall pattern of function and derivative evaluations in a standard quasi-Newton algorithm. Each line search requires one or more function evaluations, and is followed by one gradient evaluation. Extensive computational experience shows that the average number of function evaluations per line search is less than 1.5. Thus a typical sequence of function and gradient evaluations for three iterations of an optimization algorithm might be

$$f(x_{1,1}),\ f(x_{1,2}),\ \nabla f(x_{1,2}),\ f(x_{2,1}),\ \nabla f(x_{2,1}),\ f(x_{3,1}),\ \nabla f(x_{3,1})\ . \tag{3.2}$$

Here $x_{i,j}$ denotes the j-th point tried in the line search at the i-th iteration. (In some line search algorithms, the gradient is evaluated at a small percentage of unsuccessful points as well, but this does not affect our conclusions and so for simplicity we ignore this possibility.)

As stated previously, if the gradients in (3.2) are evaluated by finite differences, the gradient evaluation can make excellent utilization of parallelism since it involves n independent function evaluations that can be performed concurrently. However, if we assume that each function evaluation is performed by just one processor, and we use a standard line search, the function evaluations in (3.2) present a problem for parallelism. This is because in a standard line search, the selection of the next candidate point $x_{i,j+1}$ depends upon the value of $f(x_{i,j})$, and thus the evaluation of $f(x_{i,j})$ must be concluded before the evaluation of $f(x_{i,j+1})$ is begun. Thus, if we implement a standard line search method on a parallel computer and don't parallelize function evaluations, then while one processor performs a function evaluation, all the others processors will be idle.

This observation has led to the suggestion of developing new parallel line search algorithms that evaluate the objective function at multiple points concurrently. For example if there are p processors, one could pick p points along the search direction, evaluate $f(x)$ at all of them (one per processor), and if any

results in a decreased function value, use the point with the lowest function value as the next iterate. This parallel line search suggestion fully utilizes the multiple processors during the line search's function evaluations. Note also that it is the first suggestion we have discussed that actually changes the optimization method, i.e. the iterates produced, instead of just parallelizing it or possibly changing it due to finite precision effects.

But the property of fully utilizing all processors does not necessarily make a parallel line search algorithm desirable. To assess whether it is, one needs to consider two issues. First, one needs to ask whether the extra evaluations are useful. That is, is the parallel line search algorithm faster on a parallel machine than a straightforward implementation of a standard line search method on the same machine? The answer to this question is probably yes for most problems. All that is needed is that the extra function evaluations in the line searches lead to some decrease in the total number of iterations required to reach the minimizer, and limited experiences indicates that this is usually so.

Secondly, one needs to consider whether there are better alternatives. In this case, an alternative is to use "speculative gradient evaluation". By this, we mean that when a function evaluation is performed in the line search, the remaining $p-1$ processors are used to evaluate $p-1$ components of the finite difference gradient at this new point. If the function value at the new point is too high and the point is rejected (as with $x_{1,1}$ above), this work is wasted unless some new way is found to use these gradient components at the rejected point. If however the function value is acceptable, as it will be roughly 60-80% of the time, then the speculative gradient evaluation has enabled us to do work on otherwise idle processors that we would otherwise have to do next. Thus, it has saved us time in the parallel implementation. To illustrate this, define a concurrent function evaluation step as one where some number between 1 and p of the processors do function evaluations while the remaining processors are idle. Then for the sequence (3.2), if $n = 25$ and $p = 16$, the number of concurrent function evaluation steps is 10 for the straightforward implementation (1 for each function evaluation and 2 for each gradient evaluation) and 7 for the speculative gradient evaluation version (1 for each combined function/partial-gradient evaluation and 1 to evaluate the remaining 10 components for each gradient).

So, to assess parallel line search algorithms, one needs to compare them with parallel speculative gradient evaluation algorithms. This can in part be done analytically. The costs of the parallel line search and speculative gradient evaluation algorithm in concurrent function evaluation steps are easy to express. (We are continuing to assume that gradient evaluation is by finite differences.) For the parallel line search algorithm, assuming the best case where an acceptable lower point is always found among the p candidates, the number of concurrent function evaluation steps is

$$\left(1+\left\lceil \frac{n}{p} \right\rceil\right) * It_p$$

where It_p is the number of iterations required by the parallel line search method. For the speculative gradient evaluation method, the number of concurrent function evaluation steps is

$$\left(\delta+\left\lceil \frac{n+1}{p} \right\rceil\right) * It_s$$

where It_s is the number of iterations required by the standard sequential line search method, and $(1+\delta)$ is the average number of function evaluations per line search. Thus the parallel line search will be superior if

$$\frac{It_p}{It_s} < \frac{\delta + \left\lceil \frac{n+1}{p} \right\rceil}{1 + \left\lfloor \frac{n}{p} \right\rfloor}. \tag{3.3}$$

For the example above ($n = 25$, $p = 16$), if $\delta = 0.3$, this would require $(It_p)/(It_s) < 0.77$. As another example, if $n = 100$, $p = 16$, and $\delta = 0.3$, the parallel line search algorithm would be superior if $(It_p)/(It_s) < 0.91$. If $p \leq n+1$, the parallel line search algorithm is superior if $(It_p)/(It_s) < (\delta+1)/2$.

Unfortunately, researchers who have proposed parallel line searches do not seem to have considered the comparison of their methods to speculative gradient methods, or to have examined whether their methods satisfy (3.3) for various problem sets and values of p. Our intuition is that it will be difficult to create parallel line search algorithms that satisfy (3.3), especially if n/p is not too large. The main reason for this hypothesis is that asymptotically, the first point tried in the standard line search at each iteration is essentially the best point along this line, and so the extra function evaluations are of little or no help. Thus, we suspect that the fairly mundane option of parallel speculative gradient evaluation may be preferable to new parallel line search algorithms, under the assumptions that function evaluations are sequential and gradient evaluations are made by finite differences. Clearly, further research is needed to assess this. The main points of this subsection are that it may be difficult to construct new parallel algorithms for generic, small to moderate sized problems that are superior to intelligent parallel implementations of standard methods, and that one must consider the alternatives when assessing new methods.

3.4 Conclusions and Discussion

The discussions in this section illustrate the limitations in the development of parallel algorithms for small to medium sized, generic optimization problems. Our opinion is that at least so far, the consideration of parallelism has not led to the development of exciting new generic optimization methods for small to medium size unconstrained optimization problems. Instead, the capabilities of parallel computers have best been utilized by parallelizing existing sequential algorithms in ways that do not affect the algorithms at the optimization level. These include parallelizing the individual function evaluations, parallelizing the linear algebraic operations, and using speculative derivative evaluations. We consider it likely that similar comments will be true for algorithms for small to medium size constrained optimization problems: the use of parallelism may not lead to the discovery and use of truly novel optimization algorithms, although one may be able to utilize parallel computation effectively by simply parallelizing standard sequential algorithms.

The above statements are not meant to imply that there hasn't been interesting research on new generic parallel algorithms for small to medium sized unconstrained optimization problems. Besides the use of parallelism in line searches, some interesting work has included new quasi-Newton methods for parallel computation ([38]) and the use of partial Hessian information ([8]). So far, however, these new methods do not appear to have had a large practical impact. There has also been interesting research on new derivative-free methods for unconstrained optimization that are especially well suited to parallel computation ([12]). This research may well have an important practical impact for the special problem class for which it is intended, namely problems with very small numbers of variables.

On the other hand, once one considers large-scale problems, there are many opportunities for the development of interesting new parallel optimization algorithms, including possibilities that superior sequential methods may be discovered through this process. The remainder of this paper illustrates this in

the contexts of nonlinear equations and global optimization. There are also examples of interesting new parallel algorithms for large-scale unconstrained optimization, and we mention one very briefly. Nash and Sofer [25,26] have developed new block truncated Newton methods that are well suited to parallelism because they utilize p gradient values at each step, as opposed to one gradient value per step in the standard truncated Newton method. The new methods are significantly different than standard truncated Newton methods: in a standard truncated Newton method, a quadratic model is minimized over a subspace that is expanded by dimension one at each inner iteration, whereas in the block method, the subspace is expanded by dimension p at each inner iteration. This results in considerably different sequences of iterates. Preliminary tests by Nash and Sofer show promising results for this approach.

4. Parallel Methods for Large Problems with Special Structure -- Block Bordered Nonlinear Equations

This section illustrates the opportunities that the consideration of parallelism can provide for the creation of new methods for large-scale optimization problems with special structure. It does this by presenting one example, algorithms for the solution of block bordered systems of nonlinear equations. The material in this section is based upon [39].

Block bordered nonlinear equations are a class of large, sparse nonlinear equations that occur in many applications including VLSI design and structural engineering. Algebraically, they have the following form. The n variables and equations are grouped into $q+1$ subsets, $x_1, ..., x_{q+1}$ and $f_1, ..., f_{q+1}$, where for each i, x_i and f_i have the same number of components. Using this notation, the nonlinear system of equations has the form

$$f_i(x_i, x_{q+1}) = 0, \qquad i = 1, ..., q \ , \tag{4.1a}$$

$$f_{q+1}(x_1, ..., x_{q+1}) = 0 \ . \tag{4.1b}$$

That is, each set of equations except the last depends only upon an ''internal'' set of variables x_i and a ''linking'' set of variables x_{q+1}, while the final ''linking'' set of equations f_{q+1} involves all the variables. A common situation where this block bordered structure arises is in partitioning equations based upon a physical grid. If the variables correspond to values at the grid points, the equations correspond to relationships between values at neighboring grid points, and one partitions the grid into subregions and introduces artificial linking variables at the boundaries of the partition, a block bordered structure results. In this case, x_i, $i=1, \cdots q$ correspond to the variables within each subregion and x_{q+1} corresponds to the artificial variables.

The Jacobian matrix of (4.1) has the form

$$\begin{bmatrix} A_1 & & & & B_1 \\ & A_2 & & & B_2 \\ & & \ddots & & \vdots \\ & & & A_q & B_q \\ C_1 & C_2 & \cdots & C_q & P \end{bmatrix} \ . \tag{4.2}$$

Here the square matrices A_i are the derivatives of f_i with respect to x_i, the square matrix P is the derivative of f_{q+1} with respect to x_{q+1}, and the rectangular matrices B_i and C_i are the derivatives of f_i with respect to x_{q+1} and f_{q+1} with respect to x_i, respectively. The matrix (4.2) is referred to as a ''block-

bordered'' matrix, from whence the problem class gets its name.

A standard way to solve (4.1) is to apply Newton's method with a line search or trust region. This results in solving a linear system of equations involving a matrix of the form (4.2) at each iteration. We will refer to this approach as the ''explicit method'', for reasons that soon will become clear. An important cost of this method is the factorization of the Jacobian at each iteration. Assuming there is no pivoting between blocks, each factorization involves the factorization of each block A_i, $i = 1, \cdots, q$, along with the modifications of B_i and C_i. These q steps can be performed independently of each other. Finally it involves the formation and factorization of the matrix

$$\hat{P} = P - \sum_{i=1}^{q} C_i A_i^{-1} B_i \ . \tag{4.3}$$

The implicit method that is discussed next can be motivated by considering the costs of the factorization of (4.2) on a parallel computer. On a parallel computer, the first q steps of the factorization, factoring the diagonal blocks A_i and modifying B_i and C_i, parallelize almost perfectly assuming the q sets of blocks are partitioned equally among the processors. The final step, assembling and factoring $\hat{P}$, can be performed on one or several processors, but is likely to be a bottleneck in either case. The implicit method attempts to do more of the first type of operation, which parallelizes almost perfectly, and less of the second type.

The practical implicit method is derived from a ''pure'' implicit approach to solving (4.1). In this approach, each of the sets of internal variables x_i, $i = 1, \cdots, q$, is made into an implicit function $x_i(x_{q+1})$ of the linking variables x_{q+1} through the equation

$$f_i \left(x_i (x_{q+1}), x_{q+1} \right) = 0 \ . \tag{4.4}$$

Then the entire system of equations is reduced to solving

$$G(x_{q+1}) = f_{q+1} \left(x_1(x_{q+1}), \ldots, x_q (x_{q+1}), x_{q+1} \right) = 0. \tag{4.5}$$

It is interesting to note that the Jacobian matrix of this system is given by (4.3). This is because in both (4.5) and the final step of the factorization of (4.2), the equations (4.1a) have been used to eliminate the variables x_i, $i = 1, \cdots, q$. The difference is that in the implicit approach, these variables have been eliminated at the nonlinear rather than at the linear level.

If one applied Newton's method directly to (4.5), one would need to solve exactly for each $x_i(x_{q+1})$ for each new value of x_{q+1}. This would not be efficient. Practical variants of the implicit method come from making an approximation to this process that finds each $x_i(x_{q+1})$ much more approximately at each step. A framework for this approach is given in Algorithm 4.1 below.

Zhang, Byrd, and Schnabel [39] investigated a number of variants of the practical implicit method given by Algorithm 4.1, and their comparison to the explicit method. They found that if one performs one ''inner'' iteration on each x_i at step 1, and the outer iteration with no correction to the x_i's, then the work per iteration is almost identical to the explicit method, but the performance is inferior. In particular, the convergence is only 2-step quadratic. If, however, one also adds the correction

$$x_i = x_i - A_i^{-1} B_i \, \Delta x_{q+1} \ , \quad i = 1, \cdots, q \tag{4.6}$$

in step 2b, then the method is identical to the explicit method. So far, this just constitutes a different derivation of Newton's method.

Algorithm 4.1 -- Framework of an Implicit Method for Block Bordered Nonlinear Equations

At each iteration:

1. Inner iterations:
   For each $i = 1, \cdots, q$, perform some iterations towards solving the nonlinear equations $f_i(x_i, x_{q+1}) = 0$, with x_i as the variable and x_{q+1} fixed.
2. Outer iteration:
   a. Calculate the Newton step for $G(x_{q+1}) = 0$, with each x_i, $i \leq i \leq q$, fixed at the value from step 1.
   b. Perform a line search over all the variables $x_1, \cdots, x_{q+1}$, possibly incorporating a correction to the steps in each of the sets of variables x_i, $i=1, \cdots q$ first.

The interesting implicit methods arise when one performs more than one inner iteration on each x_i at each iteration of Algorithm 4.1, while retaining the correction (4.6). It turns out to be preferable to use the same matrices A_i and B_i for all the inner iterations of one iteration of Algorithm 4.1. Thus the additional inner iterations are quite inexpensive because no derivative matrices need to be computed or factorized.

These practical implicit methods have several good properties with respect to parallelism. First, by performing extra inner iterations, the fraction of time that the entire algorithm spends communicating between processors on a parallel computer is reduced, because the inner iterations require no communication. Secondly, they have good theoretical properties that are consistent with the efficient use of parallelism. Zhang, Byrd, and Schnabel [39] and Feng and Schnabel [13] show that these methods retain 1-step quadratic convergence per iteration of Algorithm 4.1, that the direction used in step 2b can be guaranteed to be a descent direction on $F(x)$ if the inner iteration is monitored properly, and that this monitoring can be done in a manner that does not require communication between the processors. Feng and Schnabel also show how to address singularity of the diagonal blocks A_i in a way that is amenable to parallelization and consistent with global convergence.

Most interestingly, Zhang, Byrd, and Schnabel show that these implicit methods may have computational advantages over the explicit method on sequential and parallel computers. For example, on a circuit model with 115 variables and equations that has four sets of internal variables consisting of 24, 27, 23, and 27 variables each, and 14 linking variables, the implicit method with three inner iterations per outer iteration was 15% faster than the explicit method on a sequential computer. On a parallel computer (an Intel iPSC hypercube) with 4 processors, this implicit method was 21% faster than an efficient parallel implementation of the explicit method. The advantage on the sequential computer arises because the number of outer iterations is reduced from 20 to 12, which more than compensates for the increase in the cost per iteration from the extra inner iterations. The parallel improvement is even greater because the parallel implicit method also requires less communication per iteration than the parallel explicit method, due to the higher ratio of inner to outer iterations. As a second example, on a related circuit model with twelve diagonal blocks, the implicit method with two inner iterations per outer iteration was 19% faster

than the explicit method on a sequential computer, and the parallel implicit method was 28% faster than the parallel explicit method on an Intel hypercube with 12 processors. In this case, the number of outer iterations was reduced from 18 to 12.

While these tests are not comprehensive enough for definitive conclusions, they indicate that the new implicit methods outlined in Algorithm 4.1 may be preferable to the standard explicit method for solving block bordered nonlinear equations, on parallel and sequential computers. What we find most interesting about this is that the consideration of parallelism has led to the development of interesting new methods that may be computationally advantageous, even on sequential computers. In short, this is because the consideration of parallelism led naturally to the consideration of whether there were new ways to utilize a partitioning of the problem, and this led to the development of new methods. The new methods happened to be useful computationally; they also led to interesting challenges for theoretical analysis. We speculate that there will be other large scale optimization problems where the consideration of parallelism, particularly ways to subdivide the problem to accommodate parallelism, will lead to the formulation of interesting new methods. This is one of the interesting opportunities and challenges that parallelism presents to optimization.

5. Coarse Grain Parallel Algorithms for Large-Scale Optimization -- Global Optimization

This section illustrates some of the challenges and opportunities that arise in the design and implementation of coarse-grain parallel algorithms for large-scale optimization. By coarse-grain algorithms we mean algorithms where the basic computational units are of large and possibly irregular size. There are many problems in continuous and discrete optimization that give rise to coarse-grain algorithms, and they present interesting opportunities and challenges for parallel computation. These challenges and opportunities arise both in constructing efficient parallel algorithms and in the computer science issues associated with the implementation of these algorithms.

The global optimization problem is a good example of a problem that leads naturally to coarse-grain algorithms, and is used in this section to illustrate this problem class. In particular, this section is based upon our research in constructing parallel global optimization algorithms for generic global optimization problems with rather small numbers of variables (these are still large and expensive computations), and for problems from molecular chemistry with large numbers of variables. It draws upon material in [2-6, 32-35], as well as recent, as yet unpublished work. This section does not give a comprehensive review of this research; rather it attempts to describe this research at a level sufficient to illustrate the interesting issues that arise in conjunction with constructing coarse-grain parallel optimization algorithms.

The global optimization problem is to find the lowest minimizer of a nonlinear function $f(x)$ that may have multiple local minimizers, in some closed subregion D of R^n. In this section we will assume that the subregion D is just given by upper and lower bounds on each variable. This problem arises in many practical applications, such as the molecular configuration problems discussed later in this section. It can be very difficult to solve, for two basic reasons. First, it is very difficult, if not impossible, to find mathematical approaches that lead to efficient and reliable deterministic algorithms for solving these problems. Secondly, solving global optimization problems accurately appears to require a huge amount of computation in many practical cases. For this reason, large-scale nonlinear global optimization problems were hardly attempted until recently. This means that much of the initial algorithm development for these problems has occurred in the context of parallel computation. Therefore, as opposed to the previous

sections, it is sometimes impossible to distinguish clearly between "standard" and parallel algorithms for large-scale global optimization.

A wide variety of approaches to global optimization have been proposed, mostly in the context of solving problems with just a handful (say 2-6) of parameters. We do not survey these in this paper; for comprehensive recent surveys, see [22,31]. Rather, we start by describing one approach, the stochastic global optimization approach of Rinnooy Kan and Timmer [30], that was the starting point for our parallel global optimization methods for problems with small numbers of variables.

The method of Rinnooy Kan and Timmer [30] is basically an intelligent ''multi-start'' algorithm. In a simple multi-start algorithm, one generates a number of random points x in the feasible domain D, starts a local minimization algorithm from each, and chooses the lowest local minimizer found (within D) as the candidate global minimizer. The main inefficiencies of this approach are that it may find all the local minimizers, and that it may find some of the local minimizers many times.

The approach of Rinnooy Kan and Timmer differs from simple multi-start mainly in that local minimizations are only started from carefully selected subset of the sample points, in a way that reduces or eliminates multiple local minimizations that lead to the same local minimizer. In particular, a local minimization is started from a sample point only if it has the lowest function value among all sample points within some ''critical distance'' from it. Also, the sampling/start-point-selection/local-minimization process is applied iteratively: for some number of iterations, new sample points are chosen, the critical distance is reduced by a carefully derived formula, start points for local minimizers are chosen, and the local minimizations are performed.

It can be shown that under reasonable assumptions, the method of Rinnooy Kan and Timmer finds all the local minimizers in a finite number of iterations, but that even if one iterates forever, the number of local minimizations remains finite. That is, a main inefficiency of simple multi-start has been eliminated. Computationally, this approach appears to be efficient if the number of local minimizers is fairly small. As will be mentioned later in this section, it does not appear to be an efficient approach to large-scale global optimization by itself, but is an important component of our approach to large-scale problems. For this reason, efficient parallel variants of it are of considerable practical interest.

5.1 A Simple Parallel Global Optimization Algorithm

It is fairly easy to construct a parallel algorithm that simply "parallelizes" the sequential stochastic method of Rinnooy-Kan and Timmer, and this was done as an initial research project in parallel global optimization ([4]). A straightforward and reasonably efficient way to do this is to partition the feasible region D into p subregions, where p is the number of processors. Then at each iteration, each processor samples and selects start points from its subregion. The latter step may require communication with other processors if a sample point is the lowest in its subregion within the critical distance, but is within the critical distance of other subregions. Finally at each iteration, all the start points for local minimizations are collected and distributed among the processors. It is necessary to collect and distribute start points, rather than having each processor simply perform the local minimizations from all the start points in its subregion, because subregions may have highly varying numbers of start points and the time per local minimization may vary significantly. This parallel algorithm is outlined in Algorithm 5.1.

Algorithm 5.1 has several interesting characteristics that begin to indicate the important parallel computation issues that are the focus of much of this section. First, the pieces of the algorithm that are executed in parallel between communication or synchronization points each involve large amounts of

---

Algorithm 5.1 -- Framework of a Simple Parallel Global Optimization Algorithm

Given $f : R^n \rightarrow R$, feasible region D, p processors

Partition D into p subregions

At each iteration:

1. **Sampling :** Each processor generates the coordinates of the new random sample points in its subregion, and evaluates $f(x)$ at each new sample point.
2. **Start Point Selection :** Each processor selects a subset of the sample points in its subregion to be start points for local minimizations. A sample point is selected to be a start point if it has the lowest function value of all sample points within the "critical distance" from it. This may require communication with processors that are responsible for neighboring subregions.
3. **Local Minimizations :** When all processors have completed start point selection, one processor collects all the start points and distributes them to the processors to perform the local minimizations from them. (Distribute one start point per processor; if there are more than p start points, distribute the remaining start points to the processors as they complete their current local minimizations.) When all local minimizations are complete, decide whether to stop, if not, begin the next iteration.

---

computation: sampling and evaluating $f(x)$ at large numbers of points (typically about 100), or performing one complete local minimization. This exemplifies coarse granularity. Second, due to the coarse granularity and the synchronization before and after step 3 at each iteration, it is possible that some processors may be idle for significant amounts of time. This can occur because each local minimization step (and even each start point selection step) can take widely varying amounts of time, and also because the number of local minimizations may not be a multiple of p. Third, the algorithm puts equal sampling effort into each subregion, regardless of whether or not it appears to be a fruitful region for finding the global minimizer. This is also true of the original sequential method and is not a parallel computation issue, but once one starts thinking in terms of subregions due to the consideration of parallelism, it becomes natural to consider varying this effort.

Byrd, Dert, Rinnooy-Kan and Schnabel [4] report computational experience with Algorithm 5.1 on a parallel computer. Basically, their conclusions are consistent with the above observations. On problems with relatively small numbers of variables, the method makes fairly effective use of small numbers of processors (say 8-32). But the coarse granularity and synchronization lead to idle time on some processors, and it is clear that this effect would be more pronounced with a greater number of processors. Also, it is apparent that greater efficiency in the basic method could be achieved by varying the sampling effort per subregion based upon the problem. The algorithm discussed in Section 5.2 is motivated by these observations.

5.2 An Adaptive, Asynchronous Parallel Global Optimization Algorithm

The limitations of the simple parallel stochastic global optimization method discussed above lead to the more interesting adaptive, asynchronous parallel approach for small, generic global optimization problems that we discuss next. The framework of such an algorithm is outlined in Algorithm 5.2. In comparison to Algorithm 5.1, there are two main new goals in this approach. The first is to concentrate the sampling and minimization effort in "productive" portions of the feasible domain D, i.e. subregions that are considered most likely to contain low minimizers. This goal is equally valid for sequential or parallel algorithms, but is more natural to achieve in an algorithm where the feasible region is divided into subregions, as it is for parallel computation. It is achieved by the adaptive portion of the algorithm, which consists of dynamically identifying subregions that are deemed more likely to contain low minimizers, and concentrating the sampling effort there. The second goal is to improve the load balancing of the algorithm, that is the distribution to work among processors, to eliminate processor idle time. This goal applies only to parallel implementations. It is achieved by the asynchronous part of the algorithm.

The framework that is chosen to accommodate both the adaptive and asynchronous features of Algorithm 5.2 is one where each sampling/start-point selection step for each subregion, and each local minimization from a new starting point, is a separate task. These tasks are distributed among the processors by some scheduling scheme. The overall control of this process is an important issue that is addressed below.

The adaptive, dynamic, and asynchronous aspects of Algorithm 5.2 make it significantly different than the parallel algorithms discussed earlier in this paper. Another way of stating this is that this algorithm is an irregular, task-oriented parallel algorithm, as opposed to the parallel methods discussed previously in this paper, which are either based upon data parallelism, or upon synchronized stages where at each stage, each processor performs the same task. The characteristics of Algorithm 5.2 lead to interesting algorithmic and parallel computation issues, and indicative of one of the key points of this paper: there appear to be a number of optimization problems that lend themselves to coarse-grain, irregular, task-oriented parallelism, and there are many interesting challenges and opportunities in designing and implementing these methods.

We will discuss the algorithmic and parallel computer implementation issues associated with Algorithm 5.2 only briefly, to illustrate these challenges. On the algorithmic side, one has to decide how to make the adaptive decisions in the algorithm. For example, how does one determine which subregions should receive more, or less, attention? In [32,33] this decision was based upon the percentage of low function values in the subregion relative to the overall domain. The sampling density was modified based upon this percentage, and very productive subregions were divided into smaller subregions while very unproductive subregions were skipped at some iterations. New algorithmic procedures are also necessary due to the irregular subregion and task structure: for example, rather than trying to locate neighboring subregions and obtain information about their sample points in the start point selection step, a new, self-contained ''over-sampling'' strategy was devised. These examples indicate that the consideration of adaptive, dynamic variants of existing optimization methods, like Algorithm 5.2, leads to the emergence of new algorithmic issues.

On the parallel implementation side, a key issue is how one controls and schedules the entire asynchronous, dynamic parallel algorithm. This is an important and significant research topic by itself. Smith [32] and [34,35] investigate this topic extensively. They consider fully centralized and fully distributed scheduling strategies, and introduce a new partially distributed ''centralized mediator'' strategy. Through

---

Algorithm 5.2 -- Framework of an Adaptive, Parallel Global Optimization Algorithm

Given $f : R^n \rightarrow R$, feasible region D, p processors

Partition D into $q \geq p$ subregions

For each subregion :

1. **Sampling :** Generate the coordinates of the new random sample points in the subregion, and evaluate $f(x)$ at each new sample point.
2. **Start Point Selection :** Select a subset of the sample points to be start points for local minimizations. (A sample point is selected to be a start point if it has the lowest function value of all sample points within the "critical distance" from it; special techniques that do not require communication with other processors are used for sample points near subregion boundaries.)
3. **Adaptive Decisions :** Decide whether to split this subregion into smaller subregions, what the new density of sample points for the subregion(s) should be, and the relative priority of continuing to process this subregion. Then apply this algorithm recursively to each of the new subregions as processors become available and as its priority prescribes. (Generally this will be done after the current local minimizations in step 4.)
4. **Local Minimizations :** As processors become available, perform, on some processor, a local minimization from each start point selected in step 2.

Note : A central process generally controls termination of the entire algorithm and may also perform part or all of the scheduling of the parallel algorithm.

---

modeling, simulation, and parallel implementation, they show that the centralized mediator approach has considerable advantages in scalability over a fully centralized approach, and is very competitive with a fully distributed approach while being easier to implement. The details of these results are not important to this paper, but this topic illustrates that there are significant computer science challenges associated with implementing irregular, coarse-grained parallel algorithms. No matter what the control mechanism, there are also lower level parallel implementation issues that arise for this type of algorithm. For example, one must determine how to schedule processes to minimize data movement, what the priorities among tasks should be, and how to stop the algorithm without resorting to global synchronization.

Computational results for Algorithm 5.2 on some test problems, in a parallel environment consisting of a network of computer workstations, are given in [6,33]. They show that for problems where the local minimizers are unevenly distributed in the domain, the adaptive features of Algorithm 5.2 can lead to large improvements in efficiency over Algorithm 5.1, on sequential or parallel computers. On the other hand, the gains from the asynchronous, rather than synchronous, parallel implementation are more moderate.

These experiments and several others also illustrate several other, important lessons about these adaptive parallel global optimization algorithms. First, controlling them by a centralized scheduled

process is reasonable for small numbers of processors (say 8) but likely to be a bottleneck for larger numbers of processors (say 32 or 64). This supports the need for the scheduling research mentioned above. Second, these algorithms are difficult to program and debug on parallel computers, mainly due to their asynchronous, task-oriented nature. This points to the need for research in systems or language support for asynchronous, task-oriented parallel algorithms.

Third, while the adaptive adjustments in Algorithm 5.2 greatly improve its efficiency in solving some small-scale global optimization problems, we have found that the algorithm is still not close to being an effective method for solving the much larger and more specialized problems that we discuss next. (To the best of our knowledge, this statement is equally true for other general purpose approaches to global optimization that have been developed for small problems.) Instead, as we discuss next, larger global optimization problems appear to require considerably more focused, and possibly more problem-specific, approaches.

5.3 A Parallel, Large-Scale Global Optimization Algorithm for Molecular Configuration Problems

To illustrate some of the additional challenges and opportunities that stem from larger scale parallel global optimization, the final portion of this section briefly discusses some aspects of our work in global optimization for molecular configuration problems. The molecular configuration problem is to find the configuration of a chemical molecule or compound that has the minimum potential energy. This is believed usually to correspond to the configuration that the molecule or compound assumes in nature. Many problems whose solution is very important to scientists, including the protein folding problem, are posed in this manner.

There are several aspects that make molecular configuration problems very challenging global optimization problems. First, problems of real interest have hundreds or thousands of parameters, whereas until recently, the global optimization community was developing algorithms for problems with fewer than ten parameters. Second, typical potential energy functions have vast numbers of local minimizers; often the number is believed to be an exponential function of the number of variables. Third, there are many local minimizers whose function values are very close to the global minimizer. Finally, all the local minimizers seem to have small basins of attraction (regions in the domain from which local minimizations lead to them). These last three reasons combine to make it very difficult to find the global minimizer.

When one considers constructing global optimization algorithms for problems with so many parameters, and so many hard-to-find low local minimizers, one must address two main new difficulties in comparison to smaller problems. First, how can one explore (e.g. sample) effectively in such a huge dimensional space? Adding to this challenge is the fact that for molecular configuration problems, the function values of randomly selected configurations are often many orders of magnitude higher than the functions values of good configurations. Thus one must find efficient ways to find relatively low points throughout the domain in order to get an idea of whether a particular region is likely to contain low minimizers. Second, how can one find the lowest local minimizers efficiently? One cannot afford to find all the minimizers or even all the low ones; rather, it appears one must find a way to move from low minimizers to even lower ones efficiently. These issues indicate the need for new algorithmic approaches.

Algorithm 5.3 outlines the approach taken by [2,5] to address these issues for a class of molecular configuration problems. (In particular, this approach applies to molecular clusters; some modifications

are needed for chains such as proteins or other polymers.) It is not our intent here to justify the merits of this algorithmic approach, but just to explain it sufficiently to be able to point out some interesting issues that it illustrates for parallel optimization.

The framework used in Algorithm 5.3 bears some relation to the stochastic methods discussed earlier in this section, but there are two very significant differences. First, the algorithm has two phases, one that is used to find an initial set of fairly low local minimizers, and a second that is used to move efficiently from low local minimizers to related, lower local minimizers. The first phase is closely related to the stochastic methods discussed previously, but the second phase, which accounts for the bulk of the computational work of the algorithm in practice, is quite different and essentially deterministic. The second difference will be seen to be particularly pertinent to our discussion of interesting issues for parallelism: a key technique used in the algorithm is the consideration of small dimensional subproblems within the large dimensional problem. In the first phase, this involves sampling steps that sample on only a small subset of the variables at once (step 1b). In the local minimizer improvement phase, it involves small-scale global optimizations in which only a few variables are allowed to vary with the remainder temporarily fixed (step 2b). Note that this means that the small-scale global optimization algorithm discussed above in Algorithm 5.2 can be used as a subalgorithm at Step 2b of the large-scale Algorithm 5.3.

Results of applying the approach outlined in Algorithm 5.3 to molecular configuration problems are given in [2,3,5] as well as in some forthcoming papers. One problem class the algorithm was applied to is Lennard-Jones problems. These problems consist of simple mathematical equations that model pairwise attractive/repulsive forces in clusters of identical, spherically symmetric atoms. They are much-studied in the chemistry community, in part because the forces they involve are a crucial part of the mathematical models of the energy of more complex molecular systems, including proteins and polymers. They are also very difficult global optimization problems, with the property of having many low, hard-to-locate local minimizers that was discussed above. Special purpose algorithms have been developed for this problem class based on the known geometrical structure of the solutions. Based upon these, the global minimizers are believed to be known for all the instances with up to at least 150 atoms (450 variables) ([24,28]). Our algorithm has been tested on all the instances with up to 76 atoms (228 variables), and appears to find the best known solution in all cases (including an improved solution for 72 atoms recently discovered by [10]) and a better solution than previously discovered for 75 atoms). A second problem that the approach of Algorithm 5.3 has been applied to is the conformation of water molecules modeled by a well regarded potential energy function ([9]). In the two cases tried, 20 and 21 molecules (180, 189 variables), the energy values found by Algorithm 5.3 are far lower than those found by a previous method ([23]). All these results have been obtained on parallel computers, mainly 16 or 32 processors of an Intel iPSC/860 hypercube for the Lennard-Jones problems, and 64 processors of the Intel Delta at Caltech for the water problems.

These results indicate that parallel global optimization algorithms like the one outlined in Algorithm 5.3 have promise for solving important scientific problems. For this reason, we conclude this section by discussing some of the interesting parallel computation challenges presented by this type of algorithm.

The new challenges and opportunities in the area of parallel computation that are posed by a method like Algorithm 5.3 come mainly from the multi-level nature of the algorithm. For example in Phase II, it is possible for the algorithm to work on improving multiple configurations simultaneously, with each of these improvement steps involving the use of a small-scale global optimization algorithm. This allows for the use of two or even three levels of parallelism. For instance in our runs on a distributed

---

Algorithm 5.3 - Framework of a Parallel Global Optimization Algorithms for a Class of Molecular Configuration Problems

1. **Coarse Identification of Configurations:** On each processor:

   (a) **Sampling in Full Domain:** Randomly generate the coordinates of sample configurations in the feasible domain, and evaluate the energy at each new sample point.

   (b) **One-Atom/Molecule Sampling Improvement:** For some (low energy) sample points: Select the atom/molecule that contributes the most to the energy function value, randomly sample on new locations for this atom/molecule, replace this atom/molecule in the sample configuration with the new location that gives the lowest energy value, and repeat until energy is below a threshold value.

   (c) **Full-Dimensional Local Minimizations:** Perform a local minimization from a subset of the improved sample points. Save these local minimizers for Step 2a.

2. **Improvement of Local Minimizers:**
   On each group of processors, for some number of iterations:

   (a) **Select a Configuration (and Expand it):** From the list of full-dimensional local minimizers, select a local minimizer to improve [and expand it around its center by a fixed factor, generally 1.25-2], and the atom/molecule that contributes the most (or second most) to the energy of this configuration.

   (b) **One-Atom/Molecule Global Optimization:** Apply a parallel global optimization algorithm (on this group of processors) to the energy of the selected configuration with only the coordinates of the selected atom/molecule as variables.

   (c) **Full-Dimensional Local Minimization:** Apply a local minimization procedure, with all atoms/molecules as variables, to the lowest configurations that resulted from the one-atom/molecule global optimization (one minimization at a time per processor), and merge the new local minimizers into the list of local minimizers.

---

memory multiprocessor with 64 processors, we generally improve 16 configurations simultaneously, using a group of four processors for each configuration. This means that the second level of parallelism involves using four processors for the small-scale global optimization in step 2b, and doing four large scale local minimizations simultaneously in step 2c. (Our experience indicates that using two levels of parallelism is preferable to using all the parallelism at either one of the levels in this case: at one extreme, improving 64 configurations at once is not effective due to the nature of the search space, while at the other extreme, improving one configuration at once and using 64 processors for the small-scale global optimization is not effective since the small-scale global optimization does not effectively utilize this much parallelism.) If we were using more processors, say 256, we could use a third level of parallelism at step 2c, for instance using four processors for each of the four local minimizations for each of 16 configurations.

We expect that algorithms that utilize multiple levels of parallelism will arise naturally in a number of large-scale optimization contexts, as well as many other areas of numerical computation. There is very limited experience with multi-level parallel algorithms so far in any context, and many interesting research issues arise in implementing them. For example, the issues of control and scheduling that were mentioned above for Algorithm 5.2 are even more difficult for these methods. In our two-level parallel implementation of Algorithm 5.3, it appears preferable to allow the algorithm to be asynchronous at both levels due to the irregular computational costs of the steps, and this leads to more difficult challenges in scheduling and controlling the algorithm than for the single-level asynchronous method discussed in Section 5.2. The issues involved in implementing irregular, multi-level parallel algorithms are an interesting part of the challenge in parallel large-scale optimization.

6. Summary

This paper has attempted to point out limitations, opportunities, and challenges in parallel nonlinear optimization through a set of examples. The examples are by no means exhaustive, and have been confined mainly to our own research. But they illustrate some important points.

First, it appears that so far, the consideration of parallelism hasn't led to many significant algorithmic innovations, or new theoretical challenges, for small to medium size generic optimization problems. This may be because, as is often conjectured, the best basic optimization approaches for small to moderate size problems are inherently sequential and already known. If this is true, it may mean that parallelism will mainly lead to new implementations, as opposed to fundamentally new algorithms, for these problems.

Second, it appears that for many classes of large-scale optimization problems, the consideration of parallelism may lead to the discovery of new algorithms that may be advantageous for parallel and possibly even sequential computation. This was illustrated by block-bordered systems of nonlinear equations, where the consideration of parallelism led to the investigation of new, implicit methods. It was also illustrated by global optimization problems with moderate numbers of variables, where the consideration of parallelism led naturally to the investigation of adaptive methods that dynamically partition the domain and decide which subregions should receive more or less emphasis. In both cases, the technique that led to the investigation of new methods was the partitioning of the problem into subproblems, and this partitioning was motivated by the consideration of parallelism. Related possibilities are likely to exist for other large problems.

Third, it appears that a number of large-scale optimization problems give rise to a coarse-grain, task-oriented, irregular type of parallelism. This was illustrated by the discussion of parallel global optimization; similar characteristics are seen, for example, in many branch and bound algorithms for discrete optimization problems ([37]). These types of algorithms not only are challenging from the point of view of the development of the optimization algorithm, but also pose many new challenges on the computer science side. These include the control and scheduling of the algorithms, and the development of communication and language features that make them easier to program and debug.

As a final comment, this paper has not considered problems where the optimization algorithm and the evaluation of the objective function or the constraints are combined. For example, this has been considered by [16] and others in cases where the evaluation of the objective function involves the solution of a system of partial differential equations. At one extreme, which was assumed in Section 3, the solution

of this system of differential equations can be considered an atomic unit by the optimization algorithm. At another extreme, the variables within the differential equation solver can be incorporated into the optimization problem, and the solution of the differential equations considered a set of constraints to the optimization problem. This results in a very large-scale optimization problem whose variables are both the optimization parameters and the variables within the differential equations solver. This problem may have considerable exploitable structure, but may also be very difficult to solve ([16]). There are also intermediate approaches, such as splitting the domain of the differential equations into subdomains and making only the boundary points of these subdomains variables to the optimization algorithm. These approaches offer attractive possibilities for the use of parallelism. To the extent that they prove to be fruitful ways to solve these problems, they present additional interesting challenges and opportunities for optimization algorithms that are in part motivated by the consideration of parallelism.

7. References

(1) G.S. Almasi and A. Gottlieb, *Highly Parallel Computing,* Benjamin Cummings, Redwood City, CA, 1989.

(2) R.H. Byrd, T. Derby, E. Eskow, K.P.B. Oldenkamp, and R.B. Schnabel, "A new stochastic/perturbation method for large-scale global optimization and its application to water cluster problems," *Large-Scale Optimization: State of the Art,* W. Hager, D. Hearn, and P. Pardalos, eds., Kluwer Academic Publishers, 1994, pp. 71-84.

(3) R. Byrd, T. Derby, E. Eskow, K. Oldenkamp, R.B. Schnabel, and C. Triantafillou, "Parallel global optimization methods for molecular configuration problems", *Proceedings of Sixth SIAM Conference of Parallel Processing for Scientific Computation,* SIAM, Philadelphia, 1993, pp. 165-169.

(4) R.H. Byrd, C.L. Dert, A.H.G. Rinnooy Kan, and R.B. Schnabel, "Concurrent stochastic methods for global optimization", *Mathematical Programming* 46, 1990, pp. 1-30.

(5) R.H. Byrd, E. Eskow, and R.B. Schnabel, "A new large-scale global optimization method and its application to Lennard-Jones problems", University of Colorado Technical Report CS-CS-630-92, 1992.

(6) R. H. Byrd, E. Eskow, R. B. Schnabel, and S. L. Smith, "Parallel global optimization: numerical methods, dynamic scheduling methods, and applications to molecular configuration", *Parallel Computation,* B. Ford and A. Fincham, eds., Oxford University Press, 1993, pp. 187-207.

(7) R.H. Byrd, R.B. Schnabel, and G.A. Shultz, "Parallel quasi-Newton methods for unconstrained optimization," *Mathematical Programming* 42, 1988, pp. 273-306.

(8) R.H. Byrd, R.B. Schnabel, and G.A. Shultz, "Using parallel function evaluations to improve Hessian approximations for unconstrained optimization," *Annals of Operations Research* 14, 1988, pp. 167-193.

(9) D.F. Coker and R.O. Watts, "Structure and vibrational spectroscopy of the water dimer using quantum simulation", *J. Phys. Chem.* 91, 1987, pp. 2513-2518.

(10) T. Coleman, D. Shalloway, and Z. Wu, "A parallel build-up algorithm for global energy minimizations of molecular clusters using effective energy simulated annealing." Technical Report

CTC93TR130, Advanced Computing Research Institute, Cornell University, 1993.

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

(12) J.E. Dennis Jr. and V. Torczon, "Direct search methods on parallel computers, *SIAM Journal on Optimization* 1, 1991, pp. 448-474.

(13) D. Feng and R.B. Schnabel, "Globally convergent parallel algorithms for solving block bordered systems of nonlinear equations", *Optimization Methods and Software* 2, 1993, pp. 269-295.

(14) R. Fletcher, *Practical Methods of Optimization,* Second Edition, John Wiley & Sons, New York, 1987.

(15) G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu, "Fortran D language specification", Center for Research on Parallel Computation Technical Report CRPC-TR90079, 1990.

(16) P.D. Frank and G.R. Shubin, "A comparison of optimization-based approaches for a model computational aerodynamics design problem", *Journal of Computational Physics* 98, 1992, pp. 74-89.

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

(18) G. Golub and J.M. Ortega, *Scientific Computing -- An Introduction with Parallel Computing,* Academic Press, Boston, 1993.

(19) L. Grandinetti, "Factorization versus nonfactorization in quasi-Newtonian methods for differentiable optimization," Report N5, Dipartimento di Sistemi, Universitá della Calabria, 1978.

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

(21) A. Griewank, "On automatic differentiation," *Mathematical Programming: Recent Developments and Applications,* M. Iri and K. Tanabe, eds., Kluwer Academic Publishers, Tokyo, 1989, pp. 83-108.

(22) R. Horst and H. Tuy, *Global Optimization, Deterministic Approaches*, Springer Verlag, Berlin, 1992.

(23) X. Long, "Molecular dynamics simulations of clusters -- impure van der Waals and $e^- - (H_2O)_n$ systems", Ph.D. Dissertation, Department of Chemistry, University of California, San Diego, 1992.

(24) R.S. Maier, J.B. Rosen, and G.L. Xue, "A discrete-continuous algorithm for molecular energy minimization", AHPCRC Preprint 92-031, University of Minnesota, 1992.

(25) S.G. Nash and A. Sofer, "Block truncated-Newton methods for parallel optimization," *Mathematical Programming* 45, 1989, pp. 529-546.

(26) S.G. Nash and A. Sofer, "A general-purpose parallel algorithm for unconstrained optimization," *SIAM Journal on Optimization* 1, 1991, pp. 530-547.

(27) J. Nocedal, private communication.

(28) J.A. Northby, "Structure and binding of Lennard-Jones clusters: 13 $\leq N \leq$ 147", *J. Chem. Phys.* 87, 1987, pp. 6166-6177.

(29) M.J. Quinn, *Parallel Computing -- Theory and Practice,* McGraw Hill, New York, 1994.

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

(31) A.H.G. Rinnooy Kan and G.T. Timmer, "Global Optimization", *Handbooks in Operations Research and Management Science, Volume I : Optimization,* G.L Nemhauser, A.H.J. Rinnooy Kan, and M.J. Todd, eds., North-Holland, 1989, pp. 631-662.

(32) S. Smith, "Adaptive Asynchronous Parallel Algorithms in Distributed Computation," Ph.D. Thesis, University of Colorado at Boulder, 1991.

(33) S. Smith, E. Eskow, and R.B. Schnabel, "Adaptive, asynchronous stochastic global optimization for sequential and parallel computation", *Large Scale Numerical Optimization,* T. Coleman and Y. Li, eds., SIAM, Philadelphia, 1990, pp. 207-227.

(34) S. Smith and R.B. Schnabel, "Centralized and distributed dynamic scheduling for adaptive parallel algorithms", *Unstructured Scientific Computation on Scalable Multiprocessors,* P. Mehrotra, J. Saltz, and R. Voigt, eds., MIT Press, Cambridge, Mass., 1992, pp. 301-321.

(35) S. Smith and R.B. Schnabel, "Dynamic scheduling strategies for an adaptive asynchronous parallel global optimization algorithm," University of Colorado Technical Report CU-CS-625-92, 1992.

(36) G.L. Steele, Jr., "High performance Fortran: status report", in *Proceedings of Workshop on Languages, Compilers, and Run-Time Environments for Distributed Memory Multiprocessors, SIGPLAN Notices,* Jan. 1993, pp. 1-4.

(37) H.W.J.M. Trienekens, "Parallel Branch and Bound Algorithms," Ph.D. Thesis, Erasmus University, Rotterdam, The Netherlands, 1990.

(38) P.J. van Laarhoven, "Parallel variable metric methods for unconstrained optimization," *Mathematical Programming* 33, 1985, pp. 68-81.

(39) X. Zhang, R.H. Byrd, and R.B. Schnabel, "Parallel methods for solving nonlinear block bordered systems of equations," *SIAM Journal on Scientific and Statistical Computing,* 13, 1992, pp. 841-859.