DIRECT SEARCH METHODS ON PARALLEL MACHINES*

J. E. DENNIS, JR. AND VIRGINIA TORCZON†

Abstract. This paper describes an approach to constructing derivative-free algorithms for unconstrained optimization that are easy to implement on parallel machines. A special feature of this approach is the ease with which algorithms can be generated to take advantage of any number of processors and to adapt to any cost ratio of communication to function evaluation.

Numerical tests show speed-ups on two fronts. The cost of synchronization being minimal, the speed-up is almost linear with the addition of more processors, i.e., given a problem and a search strategy, the decrease in execution time is proportional to the number of processors added. Even more encouraging, however, is that different search strategies, devised to take advantage of additional (or more powerful) processors, may actually lead to dramatic improvements in the performance of the basic algorithm. Thus search strategies intended for many processors actually may generate algorithms that are better even when implemented sequentially. The key difference is that the additional processors are not used simply to enhance the performance of an inherently sequential algorithm; they are used to spur the design of ever more ambitious—and effective—search strategies.

The algorithms given here are supported by a strong convergence theorem, promising computational results on a variety of problems, and an intuitively appealing interpretation as multidirectional line search methods.

Key words. unconstrained optimization, direct search methods, multidirectional search, parallel optimization, Nelder–Mead simplex algorithm

AMS(MOS) subject classifications. 65K05, 49D30

1. Introduction. We consider the nonlinear unconstrained optimization problem

$$\min_{\mathbf{x}\in\mathbb{R}^n} f(\mathbf{x}),$$

where $f:\mathbb{R}^n\to\mathbb{R}$. We do not use any derivatives or finite differences in our search schemes. These schemes qualify as *direct search methods* since the search is driven solely by function information. We require only that f be continuous on a compact level set to prove convergence for these methods; however, to guarantee convergence to a stationary point we require that f also be continuously differentiable. These convergence results will be discussed further in §4.

Originally, our interest in direct search methods was based on the fact that there had been only limited progress in designing effective parallel optimization algorithms that could take advantage of a large number of processors over a fairly wide range of problems. Meanwhile, current predictions suggest that achieving teraflop performance by the end of the century will require machines with 8000 to 32,000 processors. Workstations with up to 1000 processors are on the drawing board. What is needed are algorithms that can be easily *scaled* to accommodate ever larger numbers of processors—as well as ever more powerful processors. We believe that the algorithms we propose in this paper constitute progress in this direction.

The simplicity of direct search methods suggested to us that they might be easily adapted to a parallel computing environment precisely because they would be more amenable to scaling. A survey of the scientific literature also revealed that at least

* This research was sponsored by Air Force Office of Scientific Research grant AFOSR–89–0363. Equipment support was provided by the Center for Research on Parallel Computation, Rice University, Houston, Texas 77251–1892.

† Department of Mathematical Sciences, Rice University, Houston, Texas 77251–1892.

one direct search method, the Nelder–Mead simplex algorithm [16], numbers among the more popular optimization methods in scientific computing. The simplicity of the direct search methods certainly explains much of their popularity. It is also true that a lack of derivatives, as well as "noise" in the function values, may preclude the use of methods that require derivatives. Thus we believe that if we can use parallelism to improve the performance of direct search methods, any improvement will be of immediate interest and possible use. Recent experiments with problems from cancer research [1], chemical engineering [8], and stability analysis for matrix computations [9] have convinced us and our users that the approach given here is a valuable addition to the optimizer's toolkit.

The purpose of this paper is to describe these parallel direct search methods in the context of our earlier work and then to give some preliminary numerical results that indicate the potential for this approach. These results suggest the merit in pursuing the further development of parallel direct search methods. One goal is to demonstrate that, in the context of direct search methods, computing additional function values at each iteration can reduce the elapsed time to completion of an algorithm in a parallel computing environment and may actually reduce the *total* number of function evaluations required to produce an acceptable solution since the number of iterations required to reach a satisfactory solution may be significantly reduced. Thus, even though we are doing more work at each iteration (by computing more function values at each iteration), we learn so much more about the function that we produce significantly better iterates and thus converge in far fewer iterations. The net effect is that even with a more ambitious search strategy, we may actually compute fewer total function values.

Our paper is organized as follows. Section 2 outlines the basic approach we have taken to develop parallel direct search methods and gives the assumptions we have made about the general parallel computing environment as well. Section 3 contains a brief description of direct search methods and the reason for our interest in them, as well as a comparison of our approach with several parallel implementations of quasi-Newton methods for solving the general unconstrained minimization problem. In §4 we review the multidirectional search algorithm, a direct search method that forms the basis for our more general parallel schemes. We also state the applicable convergence theorem from [22]. In §5, we show how to incorporate the basic multidirectional search algorithm into a core step that can then be augmented to take better advantage of the available computational resources while retaining the convergence properties of the basic algorithm. In §6 we outline the new parallel multidirectional search algorithms and discuss implementation details. In §7, we give some preliminary numerical results that demonstrate speed-up on two fronts and report our experience with some "real" problems. In §8, we close with some remarks concerning future directions for research.

2. Approach. The approach we take can be viewed as an extremely flexible multidirectional line search method that can be easily scaled to fit the number of processors available—regardless of the size of the problem to be solved. Complete use of the available processors is accomplished in two ways. First, additional search directions are introduced systematically in an order that in some sense reflects the likelihood of producing descent. In addition, the simple line search conducted along each direction is refined, with preference given to those directions that are deemed more likely to produce descent. The work involved in determining the search directions and steps is minimal; we do not need to solve any linear systems of equations. Furthermore, this approach involves a minimum of interprocessor communication (i.e., synchronization) and places few restrictions on the function to be minimized. In particular, there is no need to assume that the function evaluations are expensive in order to justify the overhead introduced by the parallelization. This added flexibility is significant. If the function evaluations must be expensive to support the cost of the synchronization, then as processors become ever faster, the range of problems that can be solved efficiently on parallel machines becomes ever smaller. This limitation will become even more of an issue as the number of processors grows since the cost of access to remote memory will become even greater. As we shall see, one of the most attractive features of the parallel direct search methods is that these methods allow us to stack function evaluations on each processor until the cost of the computation balances the cost of the communication.

Throughout this paper we will assume that $n < p$, where n is the dimension of the problem to be solved and p is the number of available processors. While it is certainly possible to use these algorithms when $p \leq n$, the more interesting results occur when, in fact, $n \ll p$ (or, more accurately, when the total number of function values computed at each iteration of the algorithm is significantly larger than the dimension of the problem to be solved).

The results we will report in §7 have been taken from an implementation on an iPSC/860, but these algorithms can be adapted to any sort of parallel computing environment. This certainly includes either distributed-memory or shared-memory multiprocessors. However, since these algorithms are both small and flexible, they also are amenable for use on transputers or even on a network of computers that may or may not have different performance characteristics. There is only one point of synchronization, so modifications to suit a particular parallel computing environment are straightforward. Furthermore, the information we require for the synchronization is so small, regardless of the search strategy we employ, that even when global communication or some other form of access to remote memory is relatively expensive, it is still easy to choose a search strategy from among those we propose for which this approach is viable. We have concentrated on multiple instruction, multiple data (MIMD) machines only because we are ultimately interested in solving problems where the function values are themselves the result of another (expensive) process, for instance, a simulation or the solution of a differential equation. We choose to treat the function evaluation routine as a "black box." However, with the proper restrictions on the function evaluation routine, the ideas presented here could also be implemented on single instruction, multiple data (SIMD) machines. Thus we have an approach that is flexible enough to be of use in a wide range of computing environments.

3. Background. The methods given here belong to the large and often-used class called direct search methods. Direct search methods are characterized by the fact that they do not use derivatives. Derivative-free schemes are more widely applicable than gradient or quasi-Newton methods. Of course, there is little doubt that derivatives, when they are available, can be used to speed up the average-case performance of nonlinear optimization algorithms, but sometimes derivative approximations are simply not practical. In some control problems the objective calculation is so expensive that finite differences are undesirable and the code involves so much branching that automatic differentiation is currently out of the question, while an expert might require weeks, or even years, to find an alternative adjoint approach to derivative approximation. Derivative-free methods are also quite robust in dealing with objective functions that can be evaluated to only a few significant digits, which precludes the use of finite-difference derivative approximations, regardless of expense.

As an indication of how popular direct search methods are with users, one need only consult the 1989 Science Citation Index [18], which lists more than 215 citations for the classic Nelder–Mead paper. Both the number of citations and the range of journals in which these citations occur has grown every year since the paper first appeared in 1965. The most cited paper in the vast literature on quasi-Newton methods appears to be the 1963 paper of Fletcher and Powell [7], which popularized what is now known as the DFP variable metric secant update. In the 1989 Science Citation Index the Fletcher–Powell paper has 114 citations. As a further indication of its popularity, we note that the Nelder–Mead simplex algorithm also appears in most commercially available software libraries. For instance, Nelder–Mead is a standard feature of such packages as NAG, IMSL, and Matlab.

Given the popularity of the Nelder–Mead simplex algorithm, our first attempt at a parallel direct search method consisted of a straightforward implementation of the Nelder–Mead simplex algorithm on an iPSC hypercube (now referred to as an iPSC/1). While we computed $n + 4$ function values simultaneously to complete all the function evaluations that could possibly be required during the course of a single iteration, the algorithm showed only a speed-up of order two, regardless of either the size of the problem or the number of available processors. Careful examination of the behavior of the *sequential* algorithm confirmed that, in fact, this was the best we could expect to do with such a naive parallel implementation since the sequential algorithm typically required only two function values per iteration. Additional experimentation led to two interesting results. The first was the development of a new direct search method, which we call multidirectional search [21], that forms the basis for the algorithms discussed here. The second was the unexpected discovery, during our numerical testing, that the Nelder–Mead simplex algorithm can converge to nonminimizers when the dimension of the problem becomes large enough [21]. This behavior occurred for all the test problems we used from the Moré, Garbow, Hillstrom problem set [12] where the dimension of the problem could be varied. In all cases, we started with a regular simplex (i.e., a simplex with edges of equal length) from the standard starting point given in [12]. This behavior occurred even on such a simple problem as

$$\min_{\mathbf{x} \in \mathbb{R}^n} f(\mathbf{x}) = \mathbf{x}^T \mathbf{x}$$

for $n \geq 16$. One of the advantages of the multidirectional search algorithm is that, unlike the Nelder–Mead simplex algorithm, it is backed by convergence theorems that, our numerical testing indicates, are borne out in practice.

Initially our implementation of the basic multidirectional search algorithm on a shared-memory multiprocessor suffered from the fact that the number of processors that could be used was tied to the dimension of the problem. The challenge, then, was to find ways to effectively use all available processors.

Our approach consists of embedding the original multidirectional search algorithm in a family of algorithms. These algorithms have a very appealing interpretation as multidirectional line search methods. The original multidirectional search algorithm [21] performs a rudimentary line search along n search directions, hence the dependence of the use of processors on the dimension of the problem. Our new approach expands upon this idea by systematically introducing line searches along new search directions while further refining the line search along the existing set of search directions. As we shall see, this approach is extremely flexible, so that even for a problem of a given dimension on a machine with a fixed number of processors, there is a family of closely related algorithms, any one of which could be implemented.

The idea for using all available processors, or *scaling* the algorithm to fit the properties of a given machine, is an idea that we have seen in other parallel optimization algorithms. The key difference is that we do not use additional processors simply to enhance the performance of an inherently sequential algorithm.

A Newton or quasi-Newton method is essentially a sequential algorithm that consists of two distinct phases. First, a *single* search direction is constructed and then a step that satisfies some notion of sufficient decrease in the objective function value is determined. These methods are inherently sequential since a step for the current iteration cannot be determined until after the search direction has been constructed, while the search direction for the next iteration cannot be ascertained until a successful step has been found.[1] There can, however, be a significant amount of work associated with each phase of the iteration. Thus, efforts to produce general, parallel Newton or quasi-Newton methods have concentrated on parallelizing the work involved in each phase [6], [3], [4], [13], [14], [15]. These efforts have been successful at accelerating the performance of Newton or quasi-Newton methods while preserving their convergence properties. However, the inherently sequential nature of Newton's method limits the number of processors that can be successfully employed since the fundamental algorithm remains essentially unchanged. These limitations arise in two ways. Either a fairly small number of processors (i.e., no more than twenty) can be used to parallelize the linear algebra involved at each iteration (but this requires the assumption that the dimension of the problem is quite large so as to offset the cost of the synchronization), or the processors can be used to calculate finite-difference approximations to the gradient and either part or all of the Hessian. Then, for a fixed number of processors, the range of the problems that can be solved is limited, both by the dimension of the problem (i.e., $O(n) < p < O(n^2)$, where p is the number of processors and n is the dimension of the problem) and by the relative cost of the function evaluations, again to offset the cost of the synchronization.

Our approach is quite different. Given the dimension of the problem to be solved, the number of available processors, and, ideally, some notion of the relative expense of the function evaluations to communication (or synchronization) costs, we have a simple initialization scheme that tailors the basic multidirectional search algorithm to fit these specifications. The result is not just that we produce a different sequence of iterates. We actually produce *different* algorithms with *different* performance characteristics. The numerical results in §7 suggest that we often generate better direct search algorithms. This improved performance is not accidental. We compute more information about the function—more than we could easily justify in a sequential computing environment—but we use *all* of the information we compute. Not too surprisingly, we often construct a better sequence of iterates.

Before proceeding to a description of the multidirectional search algorithm, we note that this is not the only direct search method we could use as a core step for our more general parallel direct search schemes. Our investigation of the theoretical properties of the multidirectional search algorithm revealed that several well-established sequential direct search methods, such as the original factorial design algorithm of

[1] Byrd, Schnabel, and Shultz [3], [4] *anticipate* the search direction for the next iteration by calculating, speculatively, the gradient associated with the trial step. The calculation is speculative in the sense that the outcome of the trial step determines whether or not their guess was correct and thus whether or not the information they have already computed can be used to calculate the new search direction.

Box [2] or the pattern search algorithm of Hooke and Jeeves [10], also have the same essential characteristics, and thus are amenable to both the convergence analysis given in [22] and the general parallelization strategy given here in §5. The reason we have chosen to use the multidirectional search algorithm as our basic algorithm is that it requires only $O(n)$ function evaluations per iteration to guarantee convergence. The factorial design algorithm also falls under the same convergence analysis but requires $O(n^2)$ function evaluations per iteration. Thus, when there is a limited number of processors available (relative to the size of the problem), the multidirectional search algorithm is more effective. Conversely, when the number of processors is quite large, it is easy to construct examples for which the multidirectional search algorithm and the factorial design algorithm of Box generate the same generalized direct search algorithm. The similarities between these and other direct search algorithms are discussed briefly in [22]; a more detailed discussion is being prepared for publication.

We now proceed with a description of our core algorithm.

4. The basic multidirectional search algorithm. An iteration of the basic multidirectional search algorithm begins with a simplex S in $\mathbb{R}^n$, with vertices $\mathbf{v}_0, \mathbf{v}_1, \cdots \mathbf{v}_n$. The best vertex $\mathbf{v}_0$ is designated to be a vertex for which $f(\mathbf{v}_0) \leq f(\mathbf{v}_j)$ for $j = 1, \cdots, n$. We now describe a complete iteration to arrive at a new simplex S_+.

The first move of the iteration is to reflect $\mathbf{v}_1, \cdots, \mathbf{v}_n$ through the best vertex $\mathbf{v}_0$. Figure 1 shows an example for $n = 2$. The reflected vertices are labeled $\mathbf{r}_1$ and $\mathbf{r}_2$.

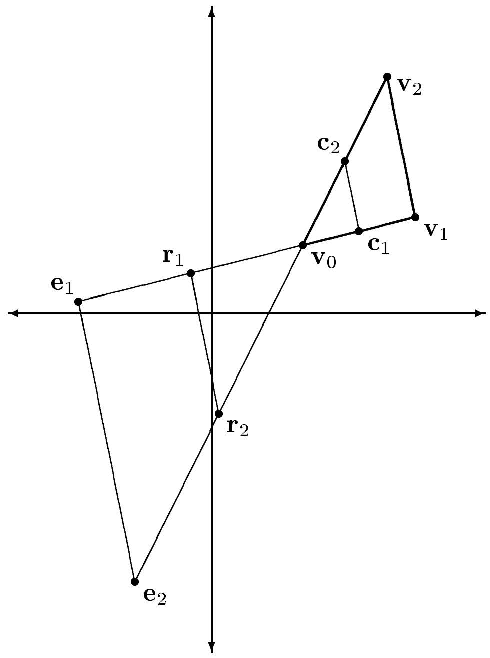

FIG. 1. *The three possible steps given the simplex S with vertices $\langle \mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2 \rangle$.*

If a reflected vertex gives a better function value than the best vertex, then the *reflection* step is called successful and the algorithm tries an *expansion* step. The expansion step consists of expanding each reflected edge $(\mathbf{r}_j - \mathbf{v}_0)$ to twice its length to give a new expansion vertex $\mathbf{e}_j$. In Fig. 1 the expansion vertices are labeled $\mathbf{e}_1$ and $\mathbf{e}_2$.

In an iteration of this basic algorithm, the expansion step would be tried only if the reflection step was successful, and it would be taken only if some expansion vertex was better than all the reflection vertices. Thus, if we try the expansion step, then the new simplex S_+ is either the expansion simplex ($\langle \mathbf{v}_0, \mathbf{e}_1, \mathbf{e}_2 \rangle$ in Fig. 1) or the reflection simplex ($\langle \mathbf{v}_0, \mathbf{r}_1, \mathbf{r}_2 \rangle$ in Fig. 1).

The other branch of the basic algorithm is the case where the reflection step was unsuccessful, i.e., no reflection vertex has a better function value than $f(\mathbf{v}_0)$. In this case, we take S_+ to be the contraction simplex formed by replacing each vertex of the worst n-face in the original simplex by the point midway from it to the best vertex. Thus, in Fig. 1, the *contraction* step takes S_+ to be $\langle \mathbf{v}_0, \mathbf{c}_1, \mathbf{c}_2 \rangle$.

To complete one iteration of the basic algorithm, we take $\mathbf{v}_0^+$ to be the best vertex of S_+.

Before giving the convergence result, we point out the line search flavor of the algorithm. In the case $n = 1$, we first try a step of a given length away from the vertex with the larger function value (the reflection step). If that is successful, then we try a longer step (the expansion step), but if it is not, then we try a step only half as long in the other orientation (the contraction step). If none of the steps give decrease, then the next iteration begins with a step in the same direction as the previous iteration began with, but only half as long. Thus, an unsuccessful sequence of iterations generates a backtracking line search with alternating orientations. (See Fig. 2.) This simple observation forms an important part of the convergence proof.

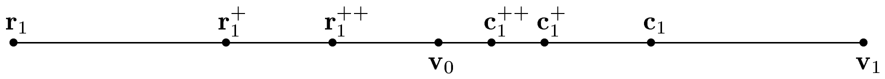

FIG. 2. *A simple backtracking line search.*

The following notation will be used in the statement of the convergence result. Let $\{\mathbf{v}_0^k\}$ be the sequence of best vertices. Let $\mathbf{v}_0^0$ be the best vertex of the simplex S_0 used to start the algorithm. Define the level set of f at $\mathbf{v}_0^0$ to be

$$L(\mathbf{v}_0^0) = \{\mathbf{x} : f(\mathbf{x}) \le f(\mathbf{v}_0^0)\}.$$

Given $\mathbf{y} \in \mathbb{R}^n$, let the contour $C(\mathbf{y})$ be

$$C(\mathbf{y}) = \{\mathbf{x} : f(\mathbf{x}) = f(\mathbf{y})\}.$$

Let X_* be the set of stationary points of the function f in $L(\mathbf{v}_0^0)$.

THEOREM 4.1. *Assume that $L(\mathbf{v}_0^0)$ is compact and that f is continuously differentiable on $L(\mathbf{v}_0^0)$. Then some subsequence of $\{\mathbf{v}_0^k\}$ converges to a point $\mathbf{x}_* \in X_*$. Thus, $\{\mathbf{v}_0^k\}$ converges to $C_* = C(\mathbf{x}_*)$ in the sense that*

$$\lim_{k\to\infty} \left[\inf_{\mathbf{x}\in C_*} \left\| \mathbf{v}_0^k - \mathbf{x} \right\| \right] = 0.$$

The assumption that f is continuously differentiable on $L(\mathbf{v}_0^0)$ can be reduced to the assumption that f is continuous on $L(\mathbf{v}_0^0)$; however, the set X_* must then be expanded to include all points where the function f is nondifferentiable on $L(\mathbf{v}_0^0)$ and where the gradient of f exists but is not continuous. The proof for both results is given in [22]. Before we go on to extend the multidirectional search algorithm to generate a family of parallel algorithms, let us discuss the proof in a form that extends to the algorithms of the next section.

First, we see why the algorithm cannot stall at a $\mathbf{v}_0$ with a nonzero gradient. Note that the edges of S adjacent to $\mathbf{v}_0$ form a basis for $\mathbb{R}^n$ and so at least one edge is not orthogonal to $\nabla f(\mathbf{v}_0)$. Thus, either that edge or its reflection is a descent direction from $\mathbf{v}_0$. Let us call this a descent edge. Now, the only way the algorithm can stay at $\mathbf{v}_0$ is to take an infinite sequence of successive contraction steps. However, at the next iteration, the contraction simplex flips to the opposite orientation to form the reflection simplex and so, along any descent edge from $\mathbf{v}_0$, we are generating a pair of sequences of points, one from each orientation, halving the distance from $\mathbf{v}_0$ at each term of the sequence, as seen in Fig. 2. Therefore, we will eventually get either a successful reflection step or a contraction step that replaces $\mathbf{v}_0$.

Thus, the algorithm can be viewed as a backtracking line search method where at least one of the n search directions is guaranteed to produce descent if $\nabla f(\mathbf{v}_0) \neq 0$. Convergence would be clear if some principle of sufficient decrease on f were required. However, we accept a new best vertex based only on simple decrease. The remainder of the proof consists of an unusual argument by contradiction. We assume that the sequence of best vertices stays uniformly bounded away from the set of stationary points. Using this assumption, along with the compactness of the level sets, the uniform linear independence of the search directions, and the continuity of ∇f, we can show that all but a finite number of vertices generated by the algorithm must be contained in a compact set and lie on a lattice. Now, the first part of the proof showed that if the best vertex is not a stationary point of the function, then the algorithm will produce a *strictly* monotonically decreasing sequence of function values. The second part of the proof demonstrates that under the hypothesis that the sequence of best vertices stays uniformly bounded away from the the set of stationary points there is only a *finite* number of function values. Therein lies the contradiction. Thus, our hypothesis cannot hold and we have convergence to the set of stationary points.

We close by noting that as long as we preserve the backtracking line search flavor of the algorithm, the proof for the basic algorithm will extend to the parallel multidirectional search algorithms we propose.

5. The parallel multidirectional search algorithms. The strategy we will employ to define a family of direct search algorithms is very simple. We will look ahead to subsequent iterations of the algorithm until we generate a sufficient number of vertices to keep all available processors busy.

We begin by removing all the branching from the basic algorithm to obtain a *core* step. Thus, the core step consists of the union of the reflection, expansion, and contraction steps from the basic algorithm. This core step will require $3n$ independent function values at each iteration.[2] Thus, in the two-dimensional example given in Fig. 1, the core algorithm computes the function values at the six new vertices $\mathbf{r}_1$, $\mathbf{r}_2$, $\mathbf{e}_1$, $\mathbf{e}_2$, $\mathbf{c}_1$, and $\mathbf{c}_2$ simultaneously. We then choose as $\mathbf{v}_0^+$ the vertex that produces the best function value while S_+ is taken to be the simplex that produced $\mathbf{v}_0^+$. If $f(\mathbf{v}_0)$ is still the least function value, then S_+ must be the contraction simplex.

There are two points to be made. First, with the stipulation that the contraction simplex must be accepted when the core step does not produce a new best vertex, the convergence theorem still holds. Second, without the branching present in the basic

[2] We will assume for now that the number of processors, p, is greater than $3n$. As we shall demonstrate in §6.3, it is possible to remove this restriction on the minimum number of processors required. In fact, as we shall see in §7, this approach may generate better *sequential* algorithms.

algorithm, we may actually produce a different sequence of iterates. For example, our choice of S_+ might now be the expansion simplex even if it would never have been constructed in the basic algorithm. This could happen if $\min\{f(\mathbf{e}_1), \cdots, f(\mathbf{e}_n)\} < f(\mathbf{v}_0) \leq \min\{f(\mathbf{r}_1), \cdots, f(\mathbf{r}_n)\}$. Thus we have a different algorithm that may actually produce a different sequence of iterates.

We have now used $3n$ processors to compute the $3n$ new vertices and their associated function values. To take advantage of more processors, we simply continue the look-ahead to subsequent iterations. For instance, we could assume that the reflection step is accepted at the current iteration, in which case one of the n new vertices associated with the reflection simplex will become $\mathbf{v}_0^+$. Thus, we can consider the new reflection vertices that might be constructed at the next iteration if each of $\mathbf{r}_1, \cdots, \mathbf{r}_n$ were given the role of $\mathbf{v}_0^+$. (See Fig. 3.) We can continue this look-ahead to construct all the reflection simplices that could be considered at the next iteration if any of $\mathbf{r}_1$, $\cdots$, $\mathbf{r}_n$, $\mathbf{e}_1$, $\cdots$, $\mathbf{e}_n$, $\mathbf{c}_1$, $\cdots$, $\mathbf{c}_n$, or $\mathbf{v}_0$ were to become $\mathbf{v}_0^+$, as shown in Fig. 4. There is also nothing to prevent us from including all possible expansion and contraction vertices as well, as seen in Fig. 5.

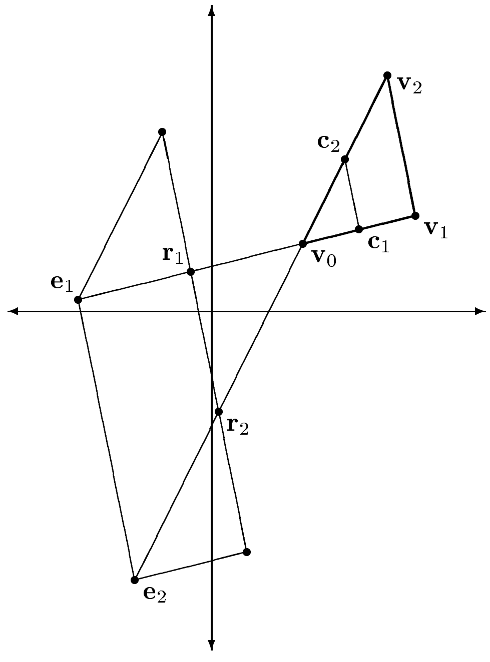

FIG. 3. *The core step with reflection steps from* $\mathbf{r}_1$ *and* $\mathbf{r}_2$.

If we were to construct all the new vertices shown in Fig. 5 and compute their associated function values in parallel, then we would have effectively completed two iterations of the basic multidirectional search algorithm. However, the numerical results we will show in §7 suggest that, in fact, we typically do much better, for reasons we will now explain.

If we return to the core step, we see that all the vertices we constructed lie along n directions determined by the n edges adjacent to $\mathbf{v}_0$, as seen in Fig. 6. We can view the core step as a rudimentary line search consisting of three steps along each of n directions. When we proceed to the next iteration and consider all possible reflection simplices, we introduce new line searches but we also further refine the search along the

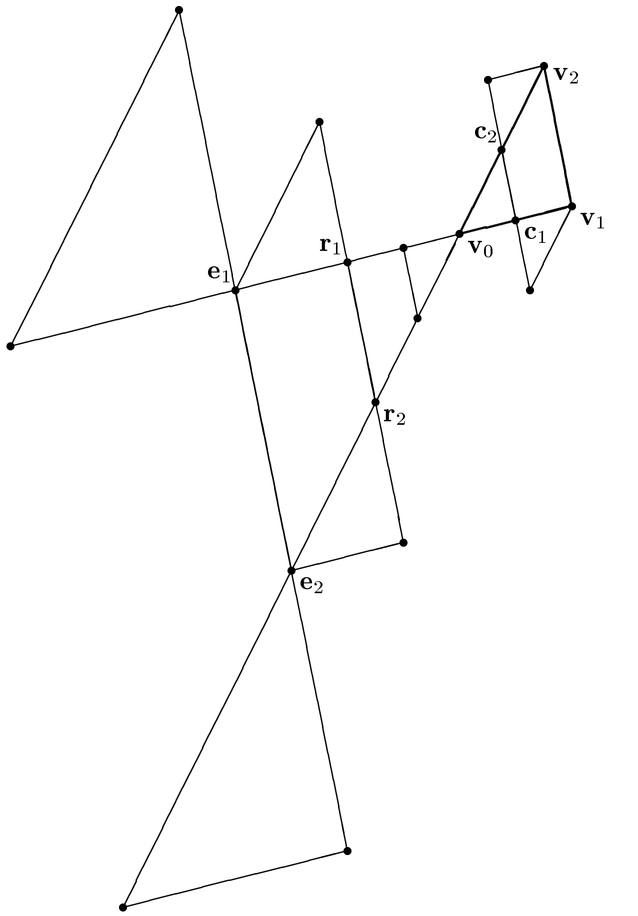

FIG. 4. *The core step with all the reflection steps for the next iteration.*

original n directions, as can be seen in Fig. 7. When we complete the full look-ahead, Fig. 8 shows that for the example we have constructed we have introduced five new line searches, each consisting of three steps, while simultaneously adding additional steps along the two original search directions to refine the line search. If we were to continue this process into the next iteration we would find that we would again introduce new line searches, we would begin to refine the search along the directions introduced in the previous iteration, and we would further refine the searches along the original n directions.

Figure 8 suggests that we have a true multidirectional line search method that scales the algorithm by introducing new line searches in a systematic way. The original n search directions are deemed most likely to produce descent since we search along edges from vertices with higher function values towards a vertex with a lower function value. In fact, the convergence theorem guarantees that if $\mathbf{v}_0$ is not a stationary point, then one of these n edges will produce descent. However, we hedge our bets by also adding new, less likely, search directions. Throughout this process we continue to refine the line search along each of the directions we have introduced, with priority going to the more likely directions.

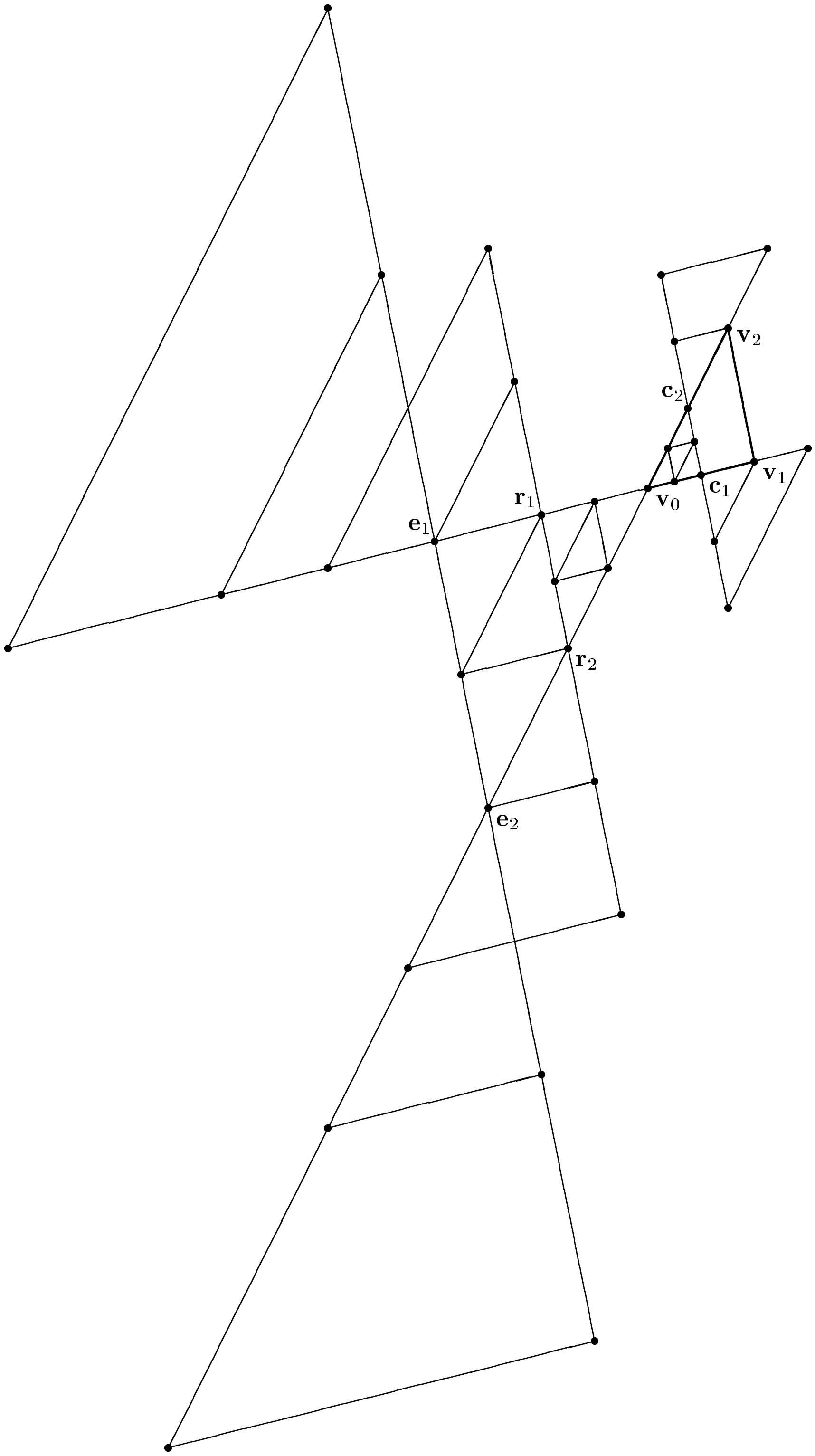

FIG. 5. *The core step with one complete look-ahead.*

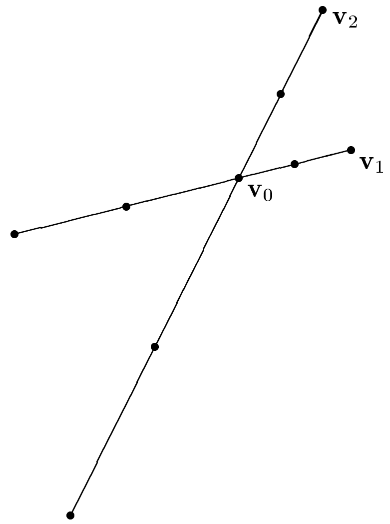

FIG. 6. *The n original search directions.*

We prefer to interpret our direct search methods as multidirectional line search algorithms. The simplex interpretation is useful for generating and programming these algorithms, as we shall see in the next section; however, the line search interpretation allows us to pose these algorithms as gradient-related methods, which helps explain both the convergence theory and the performance characteristics of these algorithms.

We also note that we have introduced a *family* of algorithms, not a single method. We have specified a scheme for introducing search directions and refining line searches, but there is tremendous flexibility within this scheme. In the example we have shown in Fig. 8, we have introduced 33 new vertices. On a 32 processor machine we can choose almost any subset consisting of 32 of these 33 vertices to define a multi-directional search algorithm. While each of these subsets was generated using the same look-ahead scheme, each subset produces a distinct algorithm that may generate a different sequence of iterates when applied to identical problems. This observation suggests the need for defining strategies to specify the order in which to introduce search directions and refine line searches. The only limitation we impose arises from the convergence theorem: we must ensure that the backtracking line search, which constructs steps along *both* orientations of the search edge, is preserved.

Defining a strategy is not difficult. We used the following principles to design our current implementation of the parallel multidirectional search algorithms:

- We construct a list of vertices until we have enough vertices to assign to all the processors.
- We start this list with $\mathbf{v}_0$ as the seed. We consider each vertex in the order in which it was added to the list and generate the *complete* core step associated with that vertex. The $3n$ vertices associated with a complete core step are then added to the bottom of the list of vertices.
- We give precedence to the reflection step, then the contraction step, and then finally the expansion step when adding vertices to the list.
- We include the current best vertex, with reduced edge lengths, only as part of the contraction step, since in the basic algorithm there would be a new best vertex if we accepted either the reflection or expansion steps.

The actual algorithm we use to implement this strategy is discussed in more detail in

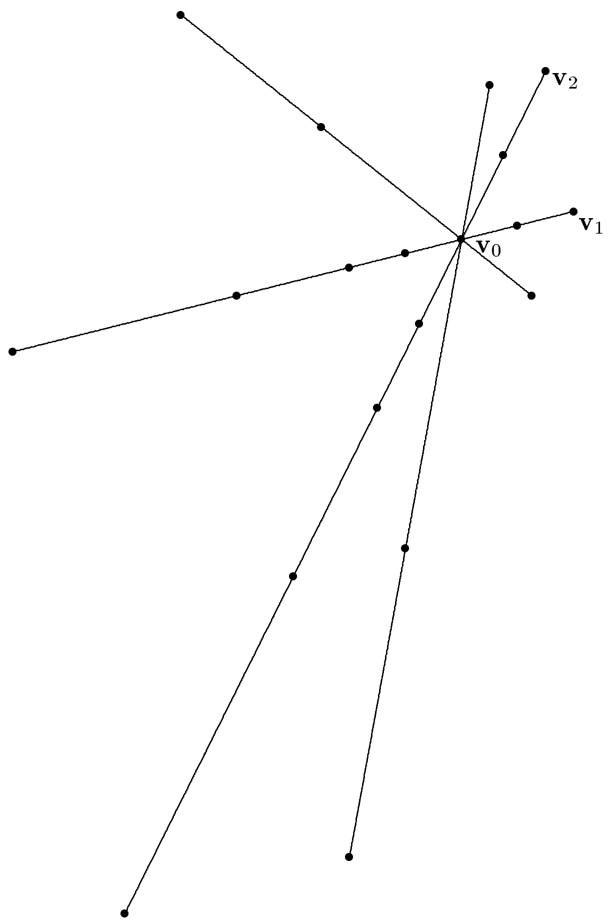

FIG. 7. *The* n *original searches with* new *steps and* additional *search directions.*

the next section.

There are certainly other, possibly better, strategies for generating parallel multi-directional search algorithms. For instance, we could first construct all the reflection vertices associated with a single iteration as in Fig. 4 and then all the contraction vertices, etc. We could also have mixed strategies that allow for different choices depending on the type of step that produced decrease in the previous iteration. These ideas are the subject of future research and will be discussed further in §8.

Another important point to note is that once we have specified a strategy for generating both the search directions and the steps, every vertex can be represented as a *fixed* linear combination of $\mathbf{v}_0$ and the edges adjacent to $\mathbf{v}_0$. There is no need to regenerate the necessary coefficients at every iteration. To see this, consider our example in Fig. 7. Since $(\mathbf{v}_1 - \mathbf{v}_0)$ and $(\mathbf{v}_2 - \mathbf{v}_0)$ span $\mathbb{R}^2$, each of the new vertices can be defined as the sum of $\mathbf{v}_0$ and a linear combination of $(\mathbf{v}_1 - \mathbf{v}_0)$ and $(\mathbf{v}_2 - \mathbf{v}_0)$. If we fix these coefficients, and then vary S, computing the new vertices at each iteration of the search reduces to computing a linear combination of the edges adjacent to the new best vertex. Thus we have a *template* for the search that is defined by our choice of strategies before the actual search procedure begins. Again, we will defer further discussion of this point to the next section.

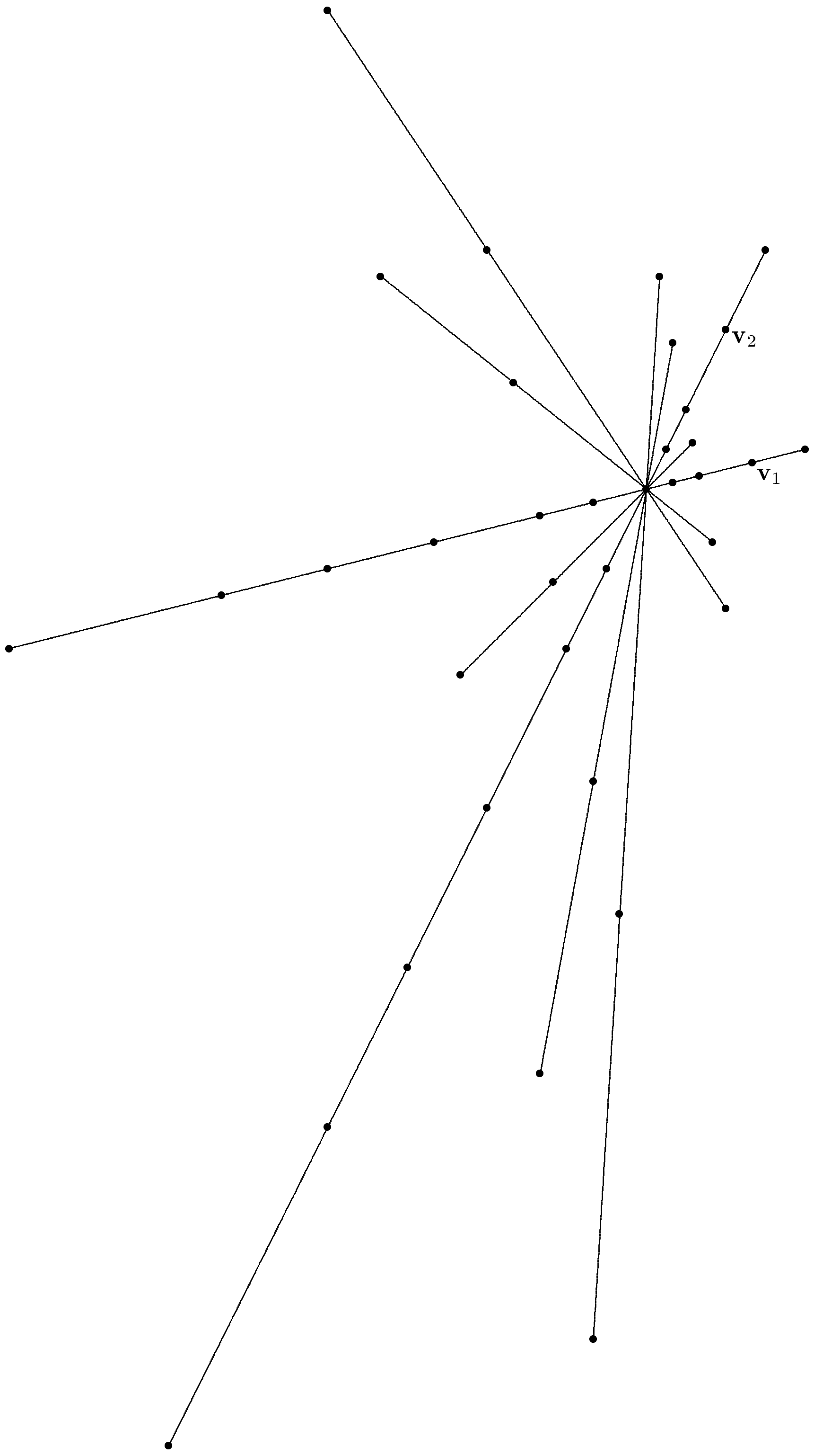

FIG. 8. *A different interpretation of the core step with one complete look-ahead.*

Another advantage of the static initialization scheme is that it allows us to eliminate duplicate vertices in the template. The reader has probably already noticed that once we begin to look ahead to subsequent iterations, some vertices may be multiply-defined. For example, in Fig. 3, $\mathbf{e}_1$ and $\mathbf{e}_2$ are redefined by the new reflection simplices. However, the coefficients necessary to construct these vertices from $(\mathbf{v}_1 - \mathbf{v}_0)$ and $(\mathbf{v}_2 - \mathbf{v}_0)$ are identical, so such duplication is easy to detect and eliminate during the initialization. The only other issue to be decided is which simplex will be associated with a multiply-defined vertex, information that is needed to avoid ambiguity when defining S_+ in the event that this vertex becomes $\mathbf{v}_0^+$. We resolve this issue by breaking ties in favor of the first simplex to define the vertex.

Having anticipated some of the major points to be addressed in any implementation of the parallel multidirectional search algorithms, we are now ready to turn to a more detailed discussion of our current implementation.

6. A distributed memory implementation. We begin with a statement of the basic algorithm, shown in Table 1. Each of the p processors[3] constructs one vertex $\mathbf{v}_i$ and its function value fv_i. The scalars $a_i, \cdots, z_i$ required to construct the vertex are local to the processor. We assume that the bulk of the computation occurs in the evaluation of $f(\mathbf{v}_i)$. Note that we have a single program running on each of the processors. The data, in the form of the scalars required to compute the vertex $\mathbf{v}_i$, varies on each processor. Thus we have a single program, multiple data model. With the appropriate restrictions on the function evaluation routine, which we choose to treat as a "black box," these methods could also be extended to SIMD machines.

Given an initial simplex S_0 with vertices $\langle \mathbf{v}_0, \mathbf{v}_1, \cdots, \mathbf{v}_n \rangle$,
    initialize template
    while (stopping criterion is not satisfied) **do**
        for $i = 1, \cdots, p$ **do**
            $\mathbf{v}_i \leftarrow \mathbf{v}_0 + a_i(\mathbf{v}_1 - \mathbf{v}_0) + \cdots + z_i(\mathbf{v}_n - \mathbf{v}_0)$
            $fv_i \leftarrow f(\mathbf{v}_i)$
        end
        $fv_* \leftarrow \min_i \{fv_i\}$ /* communication */
        update simplex
    end

TABLE 1
The parallel multidirectional search algorithm.

When each of the processors has completed its function evaluation, there is a single global exchange to determine the least function value. On the iPSC/860 we can exploit the hypercube connectivity by using a global handshake algorithm in which each processor exchanges the least function value it has seen with its nearest neighbor. If we assume a hypercube of dimension d, once each processor has exchanged information with its d nearest neighbors, every processor has fv_*. Note that this eliminates the need for a single controlling process since each processor can also test for convergence. To adapt this algorithm to a different parallel computing environment, one need only

[3] We assume, for now, enough processors to compute the $3n$ vertices associated with a core step. As we shall see in §6.3, satisfying this requirement—even when we are working on a single processor—is straightforward.

make the appropriate modifications at this single point of synchronization to introduce the appropriate form of remote memory access.

6.1. Update. During the course of the global exchange, we pass three additional pieces of information: the vertex $\mathbf{v}_*$ that produced fv_*, the pointer *source* that corresponds to the vertex in S that produced $\mathbf{v}_*$, and a scalar α_* that, in conjunction with $\mathbf{v}_*$ and *source*, allows us to construct the simplex S_* associated with $\mathbf{v}_*$. (The pointer *source* and the scalar α_i associated with the vertex $\mathbf{v}_i$ are local to each processor and are assigned during the initialization of the template.) With this additional information, updating S to produce S_+ is straightforward, as seen in Table 2. Note that before we update S, we check to see if fv_* produced the strict decrease we require. If not, then $\mathbf{v}_0$ is still the best vertex; to ensure convergence, we reduce the lengths of the edges in the simplex S by the contraction factor $\theta \in (0, 1)$.

if $(fv_* < fv_0)$ **then**
    $\mathbf{v}_0^+ \leftarrow \mathbf{v}_*$
    $\alpha_+ \leftarrow \alpha_*$
else
    $\mathbf{v}_0^+ \leftarrow \mathbf{v}_0$
    $\alpha_+ \leftarrow \theta$
    $source \leftarrow 0$
endif
for $j = 0, \cdots, n$ **do**
    $\mathbf{v}_j^+ \leftarrow \mathbf{v}_0^+ + \alpha_+(\mathbf{v}_j - \mathbf{v}_{source})$
end
swap(0,source)

TABLE 2
Updating the simplex.

6.2. Initialization. The real effort in defining a parallel multidirectional search algorithm lies in the initialization. The key point to emphasize is that this initialization is *static*. A strategy is defined before the search actually begins. The strategy can be specified as a template that is *fixed*; it is the simplex, and not the template, that varies from iteration to iteration. The template is possible because each vertex in the search scheme can be represented as a sum of $\mathbf{v}_0$ and a unique linear combination of the edges adjacent to $\mathbf{v}_0$, as demonstrated in Fig. 9. Furthermore, once we have $\mathbf{v}_0^+$, it is possible to construct S_+ from S with just two additional pieces of information, α_+ and *source*, as shown in Fig. 10.

To generate the coefficients $a_i, \cdots, z_i$ we need to construct the vertex $\mathbf{v}_i$, and the associated α_i and *source* needed to construct the simplex S_+ should $\mathbf{v}_i$ produce the least function value, we use the simple algorithm shown in Table 3. To see why this algorithm works, we begin by noting that each of the core reflection, contraction, and expansion vertices are defined as follows:

$$
\begin{aligned}
\mathbf{r}_j &= \mathbf{v}_0 + (-1)\,(\mathbf{v}_j - \mathbf{v}_0)\,, \\
\mathbf{c}_j &= \mathbf{v}_0 + \left(\frac{1}{2}\right)(\mathbf{v}_j - \mathbf{v}_0)\,, \quad \text{and} \\
\mathbf{e}_j &= \mathbf{v}_0 + (-2)\,(\mathbf{v}_j - \mathbf{v}_0)\,,
\end{aligned}
$$

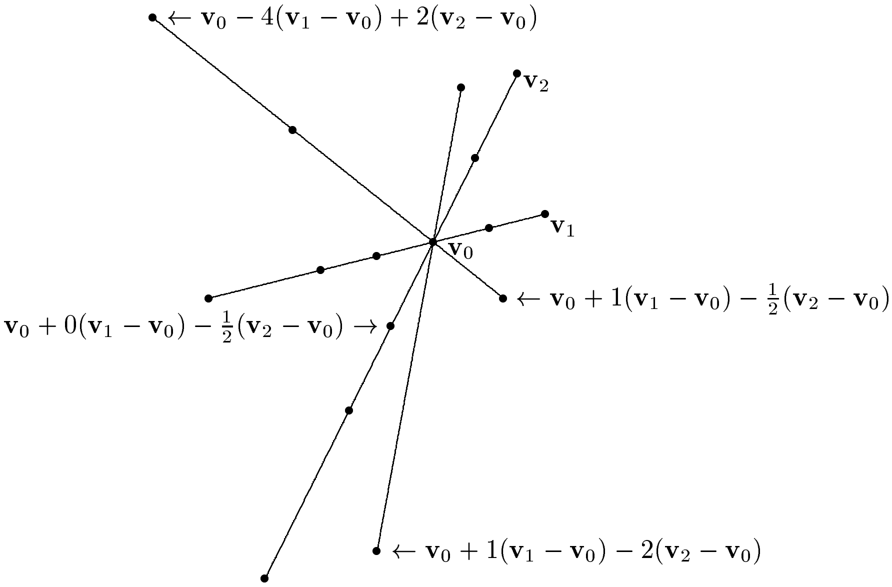

FIG. 9. *Defining the template.*

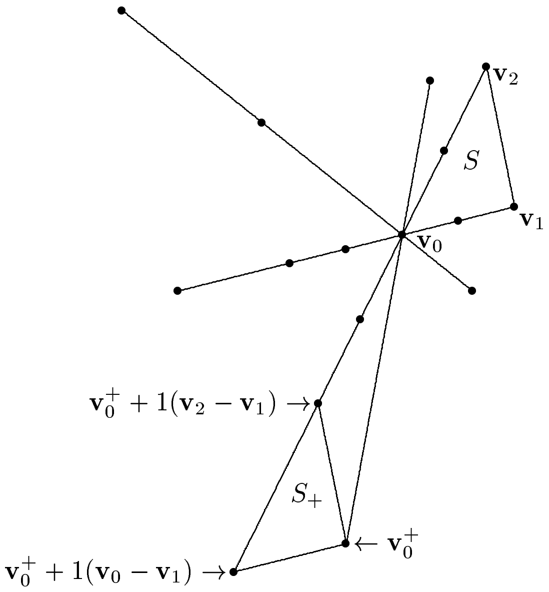

FIG. 10. *Constructing* S_+ *from* $\mathbf{v}_0^+$ *and* S *with* $\alpha_+ = 1$ *and source* $= 1$.

Given the reflection factor $\lambda = -1$, the contraction factor $\theta = \frac{1}{2}$,
and the expansion factor $\mu = -2$,
/* initialize the root of the tree by adding the current simplex */
$root \leftarrow 0$
$source_{root} \leftarrow 0$
$\alpha_{root} \leftarrow 1$
$coefficients_{root} \leftarrow \mathbf{0}$
$i \leftarrow 1$
repeat
/* generate all possible new best vertices given the current simplex */
  /* first consider all the new reflection vertices */
  for $j = 0, \cdots, n,\ j \neq source_{root}$
    $source_i \leftarrow j$
    $\alpha_i \leftarrow \lambda * \alpha_{root}$
    $coefficients_i \leftarrow coefficients_{root}$
    $coefficients_i(source_i) \leftarrow coefficients_i(source_i) + \alpha_i$
    $coefficients_i(source_{root}) \leftarrow coefficients_i(source_{root}) - \alpha_i$
    $i \leftarrow i + 1$
  end
  /* next consider all the new contraction vertices */
  for $j = 0, \cdots, n$
    $source_i \leftarrow j$
    $\alpha_i \leftarrow \theta * \alpha_{root}$
    $coefficients_i \leftarrow coefficients_{root}$
    $coefficients_i(source_i) \leftarrow coefficients_i(source_i) + \alpha_i$
    $coefficients_i(source_{root}) \leftarrow coefficients_i(source_{root}) - \alpha_i$
    $i \leftarrow i + 1$
  end
  /* finally consider all the new expansion vertices */
  for $j = 0, \cdots, n,\ j \neq source_{root}$
    $source_i \leftarrow j$
    $\alpha_i \leftarrow \mu * \alpha_{root}$
    $coefficients_i \leftarrow coefficients_{root}$
    $coefficients_i(source_i) \leftarrow coefficients_i(source_i) + \alpha_i$
    $coefficients_i(source_{root}) \leftarrow coefficients_i(source_{root}) - \alpha_i$
    $i \leftarrow i + 1$
  end
  $root \leftarrow root + 1$
until enough points have been generated

TABLE 3
Initializing the template.

for $j = 1, \cdots, n$. Note that these definitions vary only in the choice of the scalars -1, $\frac{1}{2}$, and -2 associated with each step. We also know that we can construct S_+ from S, $\mathbf{v}_0^+$, α_+, and *source* as

$$\mathbf{v}_j^+ = \mathbf{v}_0^+ + \alpha_+ \left(\mathbf{v}_j - \mathbf{v}_{source}\right),$$

for $j = 0, \cdots, n$. Thus, given $\mathbf{v}_0^+$, α_+, and *source* we can construct the core step associated with $\mathbf{v}_0^+$. For instance, the reflection vertices would be

$$\begin{aligned}
\mathbf{r}_j^+ &= \mathbf{v}_0^+ + (-1)\left(\mathbf{v}_j^+ - \mathbf{v}_0^+\right) \\
&= \mathbf{v}_0^+ + (-1)\left(\left(\mathbf{v}_0^+ + \alpha_+\left(\mathbf{v}_j - \mathbf{v}_{source}\right)\right) - \mathbf{v}_0^+\right) \\
&= \mathbf{v}_0^+ + (-1)(\alpha_+)\left(\mathbf{v}_j - \mathbf{v}_{source}\right) \\
&= \mathbf{v}_0^+ + (-1)(\alpha_+)\left((\mathbf{v}_j - \mathbf{v}_0) - (\mathbf{v}_{source} - \mathbf{v}_0)\right)
\end{aligned}$$

for $j = 0, \cdots, n$, $j \neq$ *source*—which is exactly the representation we are using to construct the template.

We also note that there is additional work to be done once the list of coefficients has been generated, since there are duplicate coefficient vectors. We simply sort the list and then eliminate duplicates. (We also include the $n+1$ vertices in the original simplex when we check for duplicates so that they are not redefined.) Preference, in terms of the α and *source* associated with each coefficient vector, goes to the first definition. Note also that if we use $\theta = \frac{1}{2}$ and $\mu = -2$ (standard choices for algorithms of this sort), then we can scale the entire procedure by an appropriately large multiple of 2 and generate the template using integer arithmetic, which is faster, halves the required storage, and eliminates round-off error.

6.3. Stacking computation. The last claim we must substantiate is that it is easy to balance the cost of the computation versus the cost of the communication, or some other form of remote memory access. When we first began testing the basic multidirectional search algorithm on a shared memory multiprocessor, memory access completely swamped any gains to be seen from computing function values—at least those from the standard test set—in parallel. One way to overcome this imbalance would be to assume that the function evaluations are expensive enough to justify the use of a parallel machine. However, this imbalance is even more acute on the iPSC/860. If we had to assume that the function evaluations are expensive in order to justify using a parallel machine, then as processors become ever faster, the class of problems we would be able to consider solving on a parallel machine would become ever smaller.

The advantage of the modification we now introduce to the parallel direct search schemes is that while it introduces more computation on the individual processors, this additional computation need not have an appreciable effect on the execution time of the algorithm. We are simply trying to balance the cost incurred due to the global communication calls. Our modification is simple. Rather than assuming that the function evaluations are expensive, we simply construct more vertices on each processor and compute their associated function values before we synchronize the search. This simple modification is given in Table 4. As we shall see in the next section, this "extra" work is not wasted; in fact, it may actually lead to a significant *decrease* in the total execution time of the algorithm since the more ambitious search strategy that results may lead to significantly fewer iterations.

We now have all the ingredients we need to claim that, given the number of processors, the dimension of the problem, and some ratio of the cost of communication

Given an initial simplex S_0 with vertices $\langle \mathbf{v}_0, \mathbf{v}_1, \cdots, \mathbf{v}_n \rangle$,
    initialize template
    while (stopping criterion is not satisfied) **do**
        for $i = 1, \cdots, p$ **do**
            for $j = 1, \cdots, k$ **do**
                $\mathbf{v}_j^i \leftarrow \mathbf{v}_0 + a_j^i(\mathbf{v}_1 - \mathbf{v}_0) + \cdots + z_j^i(\mathbf{v}_n - \mathbf{v}_0)$
                $fv_j^i \leftarrow f(\mathbf{v}_j^i)$
            end
            $fv_i \leftarrow \min_j \{fv_j^i\}$
        end
        $fv_* \leftarrow \min_i \{fv_i\}$ /* communication */
        update simplex
    end

TABLE 4
The stacked parallel multidirectional search algorithm.

to the cost of computing the function value, we can tailor a parallel multidirectional search scheme to solve the particular problem in the given computing environment. Thus we effectively remove two issues that have plagued the parallelization of other optimization algorithms: the dimension of the problem, n, and the relative expense of function evaluations.

The remaining question is then: How well do these algorithms perform? We turn to the next section for some preliminary numerical results.

7. Numerical results. We wish to demonstrate two important features of the parallel direct search methods. First, these algorithms scale almost perfectly in the sense in which "scale" is usually applied to parallel computation: if we double the number of processors we use, we essentially halve the execution time of the algorithm. In other words, we have almost perfect linear speed-up in the performance of the algorithm.

However, our parallel direct search algorithms also scale in a way that is not so usual: not only can we increase the number of processors, we can also increase the number of points we compute on each processor before we synchronize the search by making a global communication call. This is the stacked parallel multidirectional search algorithm given in Table 4. When we fix the number of processors and increase the number of points we compute on each processor we are defining a new search strategy; we have a search pattern where the number of points in the search pattern is equal to the number of processors times the number of points computed on each processor before each global communication call. When we double the number of points in the search strategy, the decrease in execution time can be dramatic. This effect on the execution time, as we shall see, is highly nonlinear; with the right choice of strategies one may even have a better *sequential* algorithm since the total number of function evaluations required to converge to a solution will be less, even if one were computing more function evaluations at each iteration.

To demonstrate these two points, we will "borrow" the level curves of a classic test problem, Rosenbrock's function [17]:

$$f(x_1, x_2) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2 \,.$$

We have borrowed the level curves because Rosenbrock's function is a function that causes most optimization problems some difficulty when the search begins at the standard starting point $(-1.2, 1)$ in the sense that fifty iterations is fairly typical of better methods.

To make the computation more realistic, particularly given the speed of the processors on the iPSC/860, we have added 10,000 extra floating point operations to each function evaluation. This is why we say we have "borrowed" the level sets of Rosenbrock's function. It is worth noting that when we added 100,000 extra floating point operations to each function evaluation the execution times did increase but the observations we now report were qualitatively the same. We chose the lesser number of extra floating point operations to demonstrate that the relative expense of the function evaluation does have an effect on the efficiency of a parallel implementation.

We tested our parallel direct search method using most of the minimization problems found in Moré, Garbow, and Hillstrom [12]. We have singled out Rosenbrock's function because it gives fairly typical results; we observed similar behavior when we experimented with other problems in the Moré, Garbow, and Hillstrom test set. Later we will discuss our experience with some "real" problems.

We started each search at the classic starting point $(-1.2, 1)$. Since we require a simplex to start the search, we used a straightforward procedure found in [11] to generate a regular simplex with edges of length one. We stopped the search when the absolute value of the function (at the best vertex $\mathbf{v}_0$) fell below 10^{-7}, a decrease of eight orders of magnitude. There is nothing special about this choice; the algorithm ran equally well for choices between 10^{-1} and 10^{-10}. (Note that the stopping test we usually employ is a little more sophisticated [21]; we resorted to this naive choice for simplicity in discussing the efficiency of the algorithm in achieving some meaningful, well-defined goal.)

We then solved the problem varying two parameters: the number of processors and the number of points in the search pattern. The results can be seen in Figs. 11 and 12.

In Fig. 11, the linear speed-up in the execution time is apparent. There are plots for six different search strategies involving search patterns of 32, 64, 128, 256, 512, and 1024 points. We have plotted the $\log_2$ of the execution time in milliseconds as a function of the number of processors we have used. If we have linear speed-up, then as we double the number of processors, the execution time should be halved. The solid lines that bracket our plots in Fig. 11 demonstrate the slope we should see for perfect linear speed-up. And, in fact, we see essentially linear speed-up for all but two of the plots. The plots for the 32 point search pattern and the 64 point search pattern do show some degradation in the speed-up when we use all 32 processors, but this is easily explained. Even with over 10,000 floating point operations per function evaluation, we are not doing enough floating point operations to offset the overhead incurred due to the communication if we are only doing one or two function evaluations per processor before each global communication call.

In Fig. 12, the nonlinear behavior associated with the *algorithmic* changes is apparent. Here there are plots for six different choices in the number of nodes used to solve the problem. We have plotted the $\log_2$ of the execution time in milliseconds as a function of the number of points in the search pattern. For this particular example the best choice is 256 points. Note, however, that a 64 point search strategy is better than a 128 point search strategy and a 512 point search strategy is better than a 1024 point search strategy. But *any* of the more ambitious strategies takes less time to

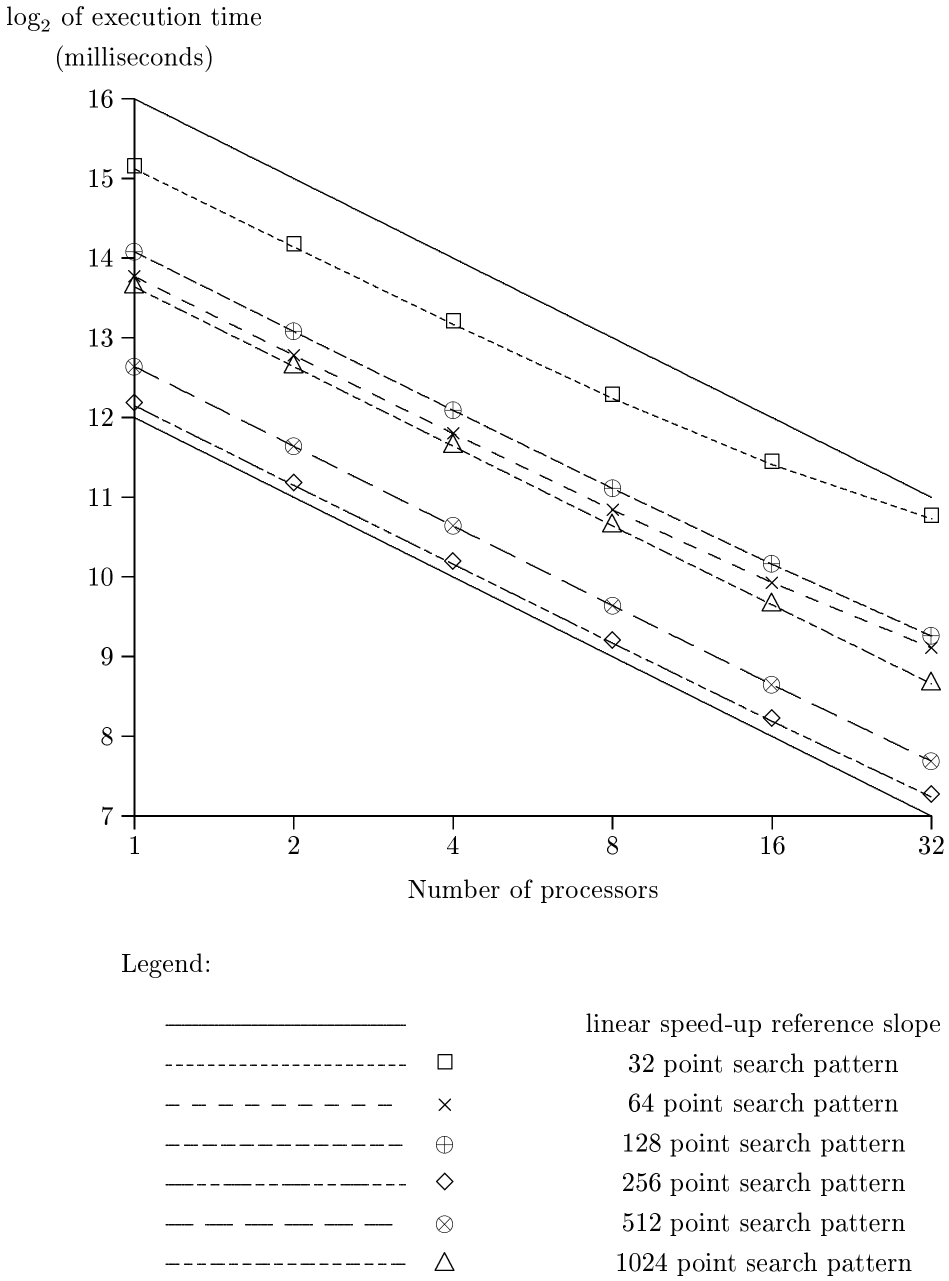

FIG. 11. *Linear speed-up obtained by doubling the number of processors.*

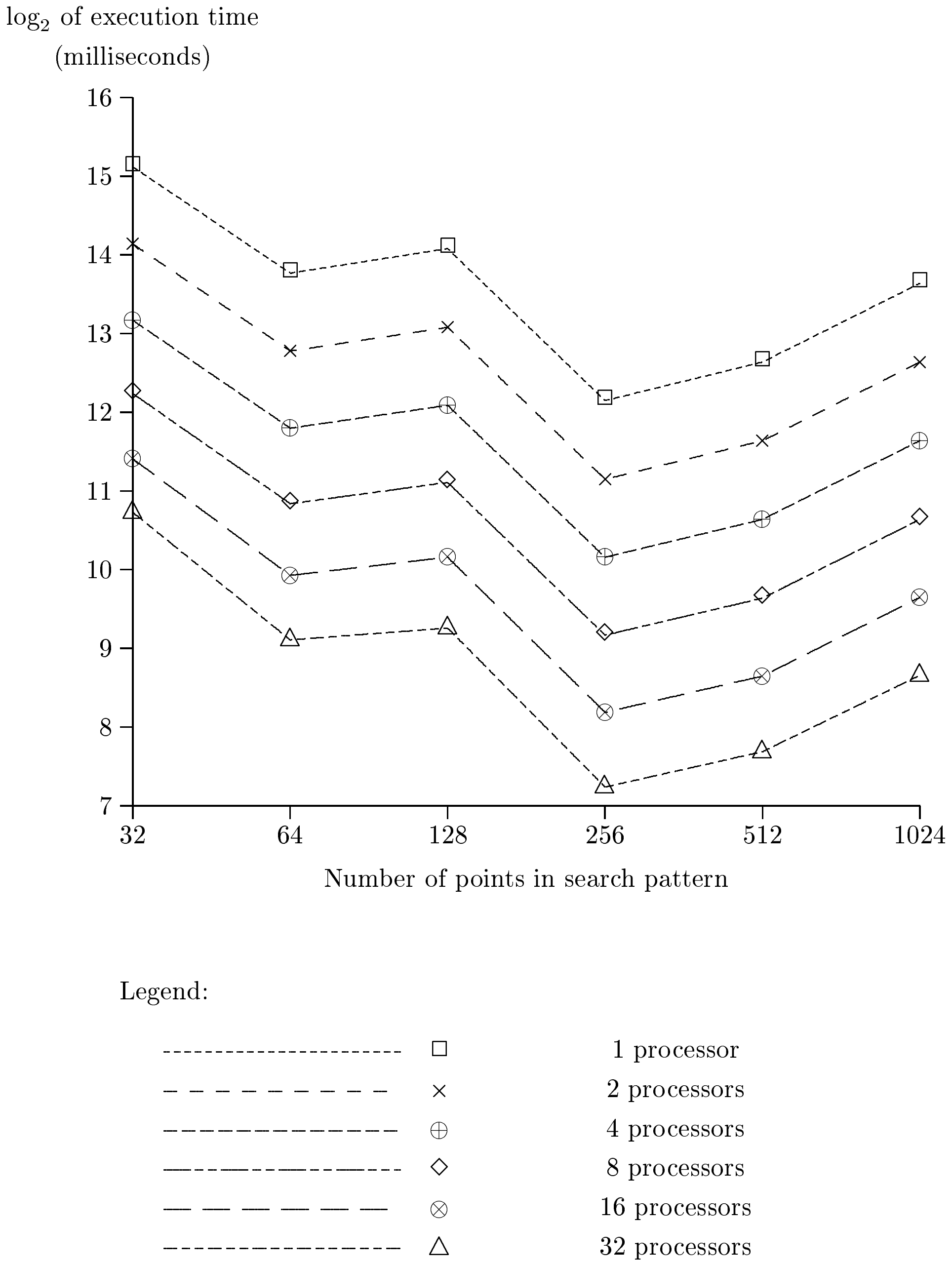

FIG. 12. *Nonlinear speed-up obtained by doubling the number of points in the search pattern.*

execute than the conservative 32 point strategy—*even on a single processor*.

The "best" choice of search strategies will vary depending on the test problem, the starting point, the size and orientation of the initial simplex, etc. It is clear that there is a certain art involved in choosing the "optimal" number of points to be used in the search pattern. However, it is also true that in general the more points used in the search strategy, the better the information obtained at each iteration and thus the better the choice of new best vertices is likely to be. Again we stress that this improvement was seen across all the problems we tested.

Furthermore, even if we do not know a priori which search strategy is "best," we can still decrease the total execution time simply by adding more processors. Returning to Fig. 12, we see that a 256 point search strategy is optimal, regardless of the number of processors we use. However, *any* of the search strategies, if implemented on at least eight processors, will execute at least as quickly as the 256 point search strategy on a single processor. Thus, even if we do not know the "optimal" search strategy, we can still expect to see a decrease in the execution time simply by increasing the number of processors we use. If we wish to solve a problem repeatedly, it may be worthwhile to spend some time identifying an "optimal" search strategy. Otherwise, we might simply use as many processors as we can find (or afford).

Given the speed of the processors on the iPSC/860, we do not even require very many processors to undertake a more ambitious strategy. We solved a parameter identification problem in three unknowns that models catalytic cracking of gasoil to gasoline [8] in just over two seconds using a 32 point search strategy on eight processors. Each function evaluation required the numerical solution of a system of ordinary differential equations. Using four processors with the same 32 point search strategy took over four seconds; two processors required between eight and nine seconds. Again, we observed the linear behavior with respect to the number of processors.

Our tests of the parallel direct search methods lead us to two conclusions. First, these algorithms do scale almost perfectly in the usual sense: as long as we require a reasonable amount of computation on each processor, the communication requirements are so minimal that we see almost perfect linear speed-up in the performance of the algorithm, regardless of the problem we solve. If we double the number of processors we halve the execution time of the algorithm.

Another result is, at present, less well understood and less predictable. If we increase the number of points in the search strategy, which often involves a significant increase in the number of function values we compute (stack) on each node before we attempt to synchronize, we may actually see a marked decrease in the total execution time of the algorithm. These improvements have been dramatic for the more difficult problems.

It is important to understand that our parallel direct search methods do *not* place an upper bound on the number of processors that can be used. Given ever more processors we expect to be able to solve ever larger problems. We also note that much of the algorithmic improvement depends on the proper choice of a search pattern and there are many different ways to generate search patterns, even for a fixed number of points. As we gain more experience with these methods, it is possible that if we can define better search strategies, then we can further increase the range of problems that can be solved efficiently.

Finally, we note that the simplicity of the parallel direct search schemes suggests that they are very useful as experimental tools. All that is needed is a function evaluation subroutine. While one is calculating the derivatives, coding a subroutine, and then debugging the code to use a more sophisticated optimization method on a "real" problem where the solution is not known, it is possible to be running experiments using the parallel direct search methods to determine reasonable starting points for the more sophisticated optimization method, as well as getting a feel for the general behavior of the function.[4]

The simplicity of the parallel direct search methods also means that they are less likely to fall prey to the pathologies, such as noise or the lack of continuity, that can plague more sophisticated optimization methods. As further evidence of this, we point to the success reported by Higham [9] using a sequential implementation, in Matlab, of the basic multidirectional search algorithm to investigate the stability and accuracy of algorithms in matrix computations. Higham observes that many of the questions of interest can be expressed in terms of some easily computable function f. The catch is that the function f is usually not smooth and derivatives, when they exist, are difficult to obtain. Thus, quasi-Newton or conjugate gradient methods are not applicable. Direct search methods, on the other hand, prove to be useful experimental tools.

We also have had experience experimenting with a data set provided by researchers at the University of Texas at Houston, M. D. Anderson Cancer Center, and the University of Texas Medical Branch at Galveston who are trying to derive mathematical models for the predictive value of early CA125 serum levels in epithelial ovarian carcinoma [1]. They have a model with five parameters that they originally fitted using NL2SOL [5]. They were concerned that NL2SOL required almost 900 function evaluations to return a solution. Experimenting with our parallel direct search methods, by successively restarting the optimization procedure, we were able to uncover that one of the parameters, which is required to be strictly greater than zero, was tending toward zero while another parameter, which is unbounded, was marching steadily towards $-\infty$—information that was not obvious from the solution returned by NL2SOL. They are now interested in further experiments to try to learn more about the behavior of their model.

The point here is not that the parallel multidirectional search algorithms produce optimal schemes for all problems on today's machines. Rather, when the problem to be solved is small, but difficult, and only a few significant digits in the solution are either required or expected, then the parallel direct search methods provide a simple and surprisingly effective approach. Furthermore, these methods may prove to be even more useful as experimental tools.

8. Future work. The next step is to try the parallel multidirectional search schemes on an inverse problem in multidimensional wave propagation. This problem can be formulated via coherency optimization as a low- (e.g., three-) dimensional minimization problem [19], [20]. Currently, an implementation of the objective function on a Stardent Titan takes, on average, several hours to return a function value. There is a tremendous amount of noise in the data so that the function values cannot be trusted to more than two digits. This means that finite-difference approximations to the gradient are really not feasible—and that an answer is expected to be accurate to only one or two digits. Thus, a quasi-Newton method seems out of the question and

[4] We had an answer for the parameter identification problem less than fifteen minutes after receiving the code for the objective function. We spent most of that time reading the accompanying instructions, compiling the code, and then entering the appropriate data. It took the iPSC/860 just over two seconds to return a solution *on our first try*. Writing a routine to evaluate the derivatives using finite-differences took thirty minutes and required some finesse. The solutions were equivalent.

a direct search method seems to be in order.

Tackling this problem, however, means that we will need to rethink the original implementation of our parallel multidirectional search schemes. To begin with, our current implementation is best suited for the case when all the function evaluations require approximately the same time to complete. Thus, there is a natural synchronization that allows us to implement the algorithm without either a controlling process or any concerns for load balancing. This will not always be the case when dealing with more difficult problems. Hence, we will need an asynchronous, task-queue-based implementation with a single controlling process.

Another direction of research would be to extend the parallel multidirectional search algorithm to problems with constraints. We believe it is possible to extend the parallel algorithms, with only minor modifications, to problems with bounded variables. We are also interested in handling linear constraints. If we can handle bounded variables, it should be possible to transfer many of the ideas learned during the development of interior point methods to our simplex-based method for handling problems with linear constraints.

There are several other ideas we would also like to pursue. Although we have a simple, fast algorithm to generate templates for the parallel multidirectional search schemes, it is possible that there are other, perhaps better, initialization schemes we could implement.

One of the few pieces of information that the basic multidirectional search algorithm carries from iteration to iteration is the size of the step taken in the previous iteration—which determines the size of the step taken in the current iteration. If an expansion step was accepted, this would indicate that the simplex is still far from a solution. If the contraction step was accepted, then either the simplex is near a solution or it is trapped in a difficult region. If we allowed mixed strategies, i.e., different templates depending on the type of step accepted in the previous iteration, then it seems possible that we could further accelerate the search. This is an idea we plan to pursue further.

Currently we place no restrictions on the simplex used to start the procedure. The default option generates a regular simplex (i.e., a simplex with edges of equal length). Another standard option would be to start with a right-angle simplex: given a starting vertex $\mathbf{v}_0$, the remaining n vertices in the simplex are generated using the following simple formula:

$$\mathbf{v}_j \leftarrow \mathbf{v}_0 + \alpha_j \mathbf{e}_j \ ,$$

for $j = 1, \cdots, n$, where α_j is a nonzero scalar and $\mathbf{e}_j$ is the standard unit coordinate vector. If we were to restrict our attention to right-angle simplices, then we would only require two n vectors to store the entire simplex. Furthermore, producing the trial vertices for the search and updating the simplex would be reduced from $O(n^2)$ operations to $O(n)$. We plan to adopt this restriction when we implement the asynchronous version of our algorithm.

There is also the possibility that if the function values are not very expensive to calculate, and the processors are very fast, then the number of function evaluations we would need to stack on each processor, k, could also become quite large. However, as we noted during our discussion of the initialization, the template is generated using integer arithmetic. The information is then rescaled before being sent out to each processor. We could, instead, store the template in its integer form, which would halve the storage requirements, and simply perform the scaling required each time the information is used. A choice then has to be made as to which is the most efficient option.

Acknowledgments. We are grateful to the referees for their useful comments. We thank Robert Michael Lewis for his valuable suggestions on how best to present this material, particularly the results given in §7.

REFERENCES

[1] E. N. Atkinson, M. G. Doherty, R. S. Freeman, and H. A. Fritsche, *Mathematical models for the predictive value of early CA125 serum levels in epithelial ovarian carcinoma.* Working paper to be submitted to *Diagnostic Oncology*, 1991.

[2] G. E. P. Box, *Evolutionary operation: A method for increasing industrial productivity*, Applied Statistics, VI (1957), pp. 81–101.

[3] R. H. Byrd, R. B. Schnabel, and G. A. Shultz, *Using parallel function evaluations to improve Hessian approximations for unconstrained optimization*, Tech. Report CS-CU-361-87, Department of Computer Science, Campus Box 430, University of Colorado, Boulder, Colorado 80309-0430, 1987.

[4] ———, *Parallel quasi-Newton methods for unconstrained optimization*, Mathematical Programming, 42 (1988), pp. 273–306.

[5] J. E. Dennis, Jr., D. M. Gay, and R. E. Welsch, *Algorithm 573 NL2SOL—an adaptive nonlinear least-squares algorithm [E4]*, ACM Transactions on Mathematical Software, 7 (1981), pp. 369–383.

[6] L. C. W. Dixon, *The place of parallel computation in numerical optimisation I, the local problem*, Tech. Report 118, Numerical Optimisation Centre, The Hatfield Polytechnic, P.O. Box 109, College Lane, Hatfield, Hertfordshire AL10 9AB, U.K., 1981.

[7] R. Fletcher and M. J. D. Powell, *A rapidly convergent descent method for minimization*, The Computer Journal, 6 (1963), pp. 163–168.

[8] G. F. Froment and K. B. Bischoff, *Chemical Reactor Analysis and Design*, John Wiley & Sons, 1979.

[9] N. J. Higham, *Optimization by direct search in matrix computations*, Tech. Report 197, Department of Mathematics, University of Manchester, Manchester, M13 9PL, U.K., 1991.

[10] R. Hooke and T. A. Jeeves, *"Direct search" solution of numerical and statistical problems*, Journal of the Association for Computing Machinery, 8 (1961), pp. 212–229.

[11] S. L. S. Jacoby, J. S. Kowalik, and J. T. Pizzo, *Iterative Methods for Nonlinear Optimization Problems*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1972.

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

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

[14] ———, *BTN: Software for parallel unconstrained optimization*, Tech. Report 64, Center for Computational Statistics, George Mason University, Fairfax, VA 22030, 1990.

[15] ———, *A general-purpose parallel algorithm for unconstrained optimization*, Tech. Report 63, Center for Computational Statistics, George Mason University, Fairfax, VA 22030, 1990. To appear in *SIAM J. Opt* (1991), v. 1, no. 4.

[16] J. A. Nelder and R. Mead, *A simplex method for function minimization*, The Computer Journal, 7 (1965), pp. 308–313.

[17] H. H. Rosenbrock, *An automatic method for finding the greatest or least value of a function*, The Computer Journal, 3 (1960), pp. 175–184.

[18] SCI, *Science Citation Index Annual 1989.* Institute for Scientific Information, Inc., Philadelphia, Pennsylvania.

[19] W. W. Symes, *Velocity inversion: A case study in infinite-dimensional optimization*, Mathematical Programming, 48 (1990), pp. 71–102.

[20] W. W. Symes and J. J. Carazzone, *Velocity inversion by coherency optimization*, Tech. Report 89-8, Department of Mathematical Sciences, Rice University, Houston, Texas 77251-1892, 1989. To appear in *Proceedings of the Workshop on Geophysics Inversion*, ed. J. B. Bednar, SIAM, 1991.

[21] V. Torczon, *Multi-Directional Search: A Direct Search Algorithm for Parallel Machines*, Ph.D. thesis, Rice University, Houston, Texas, 1989. Available as Tech. Rep. 90-7, Depart-

ment of Mathematical Sciences, Rice University, Houston, Texas 77251-1892.
[22] ———, *On the convergence of the multidirectional search algorithm*, SIAM Journal on Optimization, 1 (1991), pp. 123–145.