ASYNCHRONOUS PARALLEL PATTERN SEARCH FOR NONLINEAR OPTIMIZATION

PATRICIA D. HOUGH[†‡], TAMARA G. KOLDA[†§], AND VIRGINIA J. TORCZON[¶]

Abstract. We introduce a new asynchronous parallel pattern search (APPS). Parallel pattern search can be quite useful for engineering optimization problems characterized by a small number of variables (say, fifty or less) and by objective functions that are expensive to evaluate, such as those defined by complex simulations that can take anywhere from a few seconds to many hours to run. The target platforms for APPS are the loosely-coupled parallel systems now widely available. We exploit the algorithmic characteristics of pattern search to design variants that dynamically initiate actions solely in response to messages, rather than routinely cycling through a fixed set of steps. This gives a versatile concurrent strategy that allows us to effectively balance the computational load across all available processors. Further, it allows us to incorporate a high degree of fault tolerance with almost no additional overhead. We demonstrate the effectiveness of a preliminary implementation of APPS on both standard test problems as well as some engineering optimization problems.

Key words. asynchronous parallel optimization, pattern search, direct search, fault tolerance, distributed computing, cluster computing.

AMS subject classifications. 65K05, 90C56, 65Y05, 68W15, 90C90

1. Introduction. We consider solving the unconstrained nonlinear optimization problem, minimize $f(x)$, where $x \in \mathbb{R}^n$ and $f : \mathbb{R}^n \to \mathbb{R}$. The problems of particular interest to us are defined by computationally expensive computer simulations of complex physical processes. Such simulations may take anywhere from a few seconds to many hours of computation on a single processor. In addition, we often cannot use derivative-based methods to solve these problems because no procedure exists for the evaluation of the gradient and the function evaluations are not precise enough to produce an accurate finite-difference gradient.

Pattern search is a class of direct search methods that is popular for solving the problems described above because no derivative information is required. Further, pattern search methods admit a wide range of algorithmic possibilities; see, e.g., [15, 16, 26]. The dominant computational cost for pattern search methods lies in the evaluation of the objective function. We can exploit the definition of pattern search to derive variants that perform multiple independent function evaluations simultaneously. We then can take advantage of parallel computing platforms to reduce the overall computational cost of the search.

Both the nature of the problems of interest and the features of current distributed computing environments raise some issues that we address in this work.

The original investigation into parallel direct search methods [7, 25] made two fundamental assumptions about the parallel computing environment: 1) that the pro-

[†]Computational Sciences and Mathematics Research Department, Sandia National Laboratories, Livermore, CA 94551–9217. This research was sponsored by the Mathematical, Information, and Computational Sciences Division at the United States Department of Energy and by Sandia National Laboratory, a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE–AC04–94AL85000.

[‡]Email: pdhough@sandia.gov.

[§]Email: tgkolda@sandia.gov.

[¶]Department of Computer Science, College of William & Mary, Williamsburg, VA 23187–8795. Email: va@cs.wm.edu. This research was funded in part by the Computer Science Research Institute at Sandia National Laboratories and by the National Science Foundation under Grant No. CCR–9734044.

cessors were both homogeneous and tightly coupled and 2) that the amount of time needed to finish a single evaluation of the objective function was effectively constant. It is time to reexamine these two assumptions. Clearly, given the current variety of parallel computing platforms, including distributed systems comprising loosely-coupled, often heterogeneous, commercial off-the-shelf components [24], the first assumption is no longer valid. The second assumption may not hold in our case because we focus on problems defined by the simulations of complex physical processes. Typically, the simulations themselves are based on iterative numerical techniques and so the assumption that evaluations of the objective finish in constant computational time on equivalent processors often does not hold. In fact, the behavior of the simulation for any given input is difficult to assess in advance since it can vary substantially depending on a variety of factors.

Because the original assumptions underlying parallel direct search are not valid for the situations we now face, we can no longer assume that the computation proceeds in lockstep. A single synchronization step at the end of every iteration, as in [25], is neither appropriate nor effective when any of the following factors holds: function evaluations finish in varying amounts of time (even on equivalent processors), the processors employed in the computation possess different performance characteristics, or the processors have varying loads. Our goal is to introduce a class of asynchronous parallel pattern search (APPS) methods that make more effective use of a variety of computing environments, as well as to devise strategies that accommodate the variation in completion time for function evaluations. Our approach is outlined in §3.

Another consideration we address in this paper is incorporating fault-tolerant strategies into APPS since one intent is to use this software on large-scale systems. As the number of individual computers participating in a computation grows, the chance that one (or more) will fail also grows. If we embark on a lengthy computation, we want reasonable assurance of producing a final result, even if a subset of processors fails. Thus, our goal is to design methods that respond to such failures and protect the solution process. Rather than simply checkpointing intermediate computations to disk and then restarting in the event of a failure, we are instead considering methods with heuristics that adaptively modify the search strategy. We discuss the technical issues in further detail in §4.

In §5 we provide numerical results, for both standard and engineering optimization test problems, that compare a preliminary implementation of APPS with an implementation of parallel pattern search (PPS) that incorporates a blocking synchronization point within each iteration. Finally, in §6 we outline additional questions to pursue.

Although we are not the first to embark on the design of asynchronous parallel optimization algorithms, we are aware of little other work, particularly in the area of nonlinear programming. Approaches to developing asynchronous parallel Newton or quasi-Newton methods are proposed in [4, 10], though the assumptions underlying these approaches differ markedly from those we address. Specifically, both assume that solving the Newton equation at each iteration is the dominant computational cost of the optimization algorithm because the dimensions of the problems of interest are relatively large. A different line of inquiry [23] considers the use of quasi-Newton methods in the context of asynchronous stochastic global optimization algorithms; we only consider the problem of identifying local stationary points.

2. Parallel Pattern Search. Before proceeding to a discussion of APPS, let us first review some key features of pattern search.

A primary characteristic of pattern search methods is that they sample the function over a predefined pattern of points, all of which lie on a rational lattice. By enforcing structure on the form of the points in the pattern, as well as simple rules on both the outcome of the search and the subsequent updates, we are guaranteed global convergence to a stationary point [9, 16, 26].

For our purposes, the feature of pattern search that is amenable to parallelism is that once the candidates in the pattern have been defined, the function values at these points can be computed independently and, thus, concurrently.

To make this more concrete, consider the following particularly simple version of a parallel pattern search algorithm. At iteration k, we have an iterate $x_k \in \mathbb{R}^n$ and a step-length control parameter $\Delta_k > 0$. The pattern of p search directions is denoted by $\mathcal{D} = \{d_1, \ldots, d_p\}$. Although other choices for $\mathcal{D}$ are possible, for our simple variant we choose $\mathcal{D} \equiv \{e_1, \ldots, e_n, -e_1, \ldots, -e_n\}$, where e_j represents the jth unit vector. Fig. 2.1 illustrates an example of this search pattern when $n = 2$.

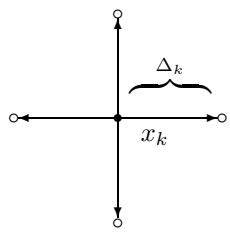

FIG. 2.1. *A simple instance of a pattern for pattern search.*

Now that we have selected $\mathcal{D}$, multiple algorithmic options are open to us. An obvious strategy for concurrent computing is to identify an $x_+ \in \{x_k + \Delta_k d_i,\ i = 1, \ldots, p\}$ such that $f(x_+) = \min\{f(x_k + \Delta_k d_i), i = 1, \ldots, p\}$. This strategy requires us to compute $f(x_k + \Delta_k d_i)$ for all p vectors in the set $\mathcal{D}$. To ensure global convergence of some subsequence to a stationary point we can accept any point $x_k + \Delta_k\, d_i$ for which $f(x_k + \Delta_k d_i) < f(x_k)$ [26]. Thus, finding $f(x_+) = \min\{f(x_k + \Delta_k d_i),\ i = 1, \ldots, p\}$ is in some sense more than is really needed. However, concurrency masks the computational expense of the stronger acceptance condition.

If none of the points in the pattern reduces the objective, then we set $x_{k+1} = x_k$ and reduce Δ by setting $\Delta_{k+1} = \frac{1}{2}\Delta_k$; otherwise, we set $\Delta_{k+1} = \Delta_k$ and $x_{k+1} = x_+$. We repeat this process until some reasonable stopping criterion, such as $\Delta_k \leq tol$, is satisfied [8, 9]. This basic strategy leads us to the algorithm we call parallel pattern search (PPS), which is given in Fig. 2.2.

There still remains the question of what constitutes an acceptable pattern. Following the examples in [16], we borrow the following definition from [6]:

DEFINITION 2.1. *A set of vectors $\{d_1, \ldots, d_p\}$ positively spans $\mathbb{R}^n$ if any vector $v \in \mathbb{R}^n$ can be written as a nonnegative linear combination of the vectors in the set; i.e., for any $v \in \mathbb{R}^n$ there exist $\alpha_1, \alpha_2, \ldots, \alpha_p \geq 0$ such that*

$$v \quad = \quad \alpha_1 d_1 + \cdots + \alpha_p d_p.$$

We require $\mathcal{D}$ to be a *positive spanning set* for $\mathbb{R}^n$. (This is a bit of a misnomer; given the definition, it perhaps would be more apt to to call it a "nonnegative" spanning set.) We add the condition that $\mathcal{D}$ be composed of rational vectors [16].

Initialization:

- Set the iteration counter $k = 0$.
- Select a set of search directions $\mathcal{D} = \{d_1, \ldots, d_p\}$.
- Select a step-length control parameter Δ_0.
- Select a stopping tolerance *tol*.
- Select a starting point x_0 and evaluate $f(x_0)$.

Iteration:

1. Compute $x_k + \Delta_k d_i$ and evaluate $f(x_k + \Delta_k d_i)$, for $i = 1, \ldots, p$, *concurrently.*
2. Determine x_+ and $f(x_+)$ such that $f(x_+) = \min \{ f(x_k + \Delta_k d_i),\ i = 1, \ldots, p \}$ *(synchronization point).*
3. If $f(x_+) < f(x_k)$, then $x_k \leftarrow x_+$ and $f(x_k) \leftarrow f(x_+)$. Else $\Delta_k \leftarrow \frac{1}{2}\Delta_k$.
4. If $\Delta_k > tol$, $k \leftarrow k + 1$, go to Step 1. Else, exit.

FIG. 2.2. *The PPS algorithm.*

3. Asynchronous Parallel Pattern Search. The appeal of the parallel pattern search strategy outlined in Fig. 2.2 is that it is straightforward to implement. Unfortunately, inefficiencies in processor utilization for PPS arise when the objective function evaluations do not finish in approximately the same amount of time. This may happen for several reasons. First, the objective function evaluations may be complex simulations that require different amounts of work depending on the input parameters. Second, the computational loads on the individual processors may vary. Third, the processors participating in the calculation may possess different computational characteristics. When the objective function evaluations take varying amounts of time, those processors that can finish their share of the computation more quickly wait for the remaining processors to contribute their results. Fourth, the number of processes we are interested in executing may not exactly match the number of available processors. Finally, there is the real risk that either processes or processors may fail during the course of the computation. For all these reasons, we pursue a more versatile concurrent strategy, which we call asynchronous parallel pattern search (APPS), that allows us to effectively balance the computational load across the available processors.

Were we simply interested in load-balancing issues, the *master-slave* paradigm for the design of parallel programs would be inviting. Given such a design perspective, we could localize all decision making to a single process (the master) and devote all remaining processes (the slaves) to the evaluation of $f(x_{\text{trial}})$ for various choices of x_{trial} determined by the master process. Since we are assuming that the evaluation of the objective is the dominant computational cost, we would not have to be overly concerned about the communication bottlenecks that can sometimes occur using such a paradigm. However, fault tolerance is our other prominent concern. If we localize the decision-making to a single process and the master process fails, we would be unable to finish the computation. (Recovery in the event that one of the slave processes fails is easy; once a failure is detected, the master process can simply restart the failed slave process.)

Such concerns lead us to the *peer-to-peer* paradigm. We want each process to be an independent unit, capable of making its own decisions and equipped to respond intelligently whenever it detects that other processes have failed.

3.1. Peer-to-Peer Synchronous PPS. In order to better understand APPS, let us first consider a peer-to-peer version of synchronous PPS.

For parallel pattern search, there are p processes, with each process in charge of a single search direction in the set $\mathcal{D}$. In Fig. 3.1, we show the peer-to-peer version

Initialization:

- Determine my search direction $d \in \mathcal{D}$.
- Receive the initial value of the step-length control parameter Δ_{trial}.
- Receive the value of the stopping tolerance tol.
- Receive the starting point x_{best} and receive (or evaluate) $f_{\text{best}} \equiv f(x_{\text{best}})$.

Iteration:

1. Compute $x_{\text{trial}} \leftarrow x_{\text{best}} + \Delta_{\text{trial}}\, d$ and evaluate $f_{\text{trial}} \equiv f(x_{\text{trial}})$.
2. Perform a *global reduction* to determine f_+ (and the associated x_+) such that f_+ is the minimum of all f_{trial} values on all p processes.
3. (a) If $f_+ < f_{\text{best}}$, then
   i. $\Delta_{\text{trial}} \leftarrow \lambda\, \Delta_{\text{trial}}$ with $\lambda \in \{2^\ell \mid \ell \in \mathbb{Z}_+\}$ and
   ii. $\{x_{\text{best}}, f_{\text{best}}\} \leftarrow \{x_+, f_+\}$.
   (b) Else $\Delta_{\text{trial}} \leftarrow \frac{1}{2}\Delta_{\text{trial}}$.
4. If $\Delta_{\text{trial}} > tol$, go to Step 1. Else, exit.

FIG. 3.1. *Peer-to-peer version of (synchronous) PPS.*

of synchronous PPS from the perspective of a single process. We drop the subscript $i = \{1, \ldots, p\}$ to emphasize that each process is only concerned with its own unique direction. In the initialization, each process determines its search direction and "receives" the values of Δ_{trial}, tol, and x_{best} either by reading them from an input file or by receiving them in a message from another process.

In the main iteration of PPS, the only communication a process has with its peers is the reduction in Step 2, where all the processes participating in the computation contribute their values for x_{trial} and f_{trial}. The reduction operation returns f_+, the minimum value of f_{trial} over all processes, and x_+, the associated point. This reduction operation is the synchronization point for PPS—the minimum value of f_{trial} over all processes cannot be determined until all processes have finished their evaluation of $f(x_{\text{trial}})$.

As indicated in Step 3ai, we may increase Δ_{trial} when a decrease in f is obtained. We have two possible reasons for doing so. First, we do not want Δ_{trial} to become too small based on the outcome of a search along a single direction. So if we find a step that produces decrease in f, but for which Δ_{trial} is smaller than some $\Delta_{\min} > tol$, then we choose the least nonnegative integer ℓ (i.e., $\ell \in \mathbb{Z}_+$) such that $2^\ell \Delta_{\text{trial}} > \Delta_{\min}$ (in our implementation we somewhat arbitrarily choose $\Delta_{\min} \equiv 2^3 \cdot tol$). Assuming, instead, that we ended the search successfully with a choice of Δ_{trial} that satisfies $\Delta_{\text{trial}} > \Delta_{\min}$, we may still choose to expand Δ_{trial}. In our implementation we double Δ_{trial} (i.e., we choose $\ell = 1$) if the same search direction produces at least two successful iterates in a row. Our reason for this condition is straightforward: if we have just completed a sequence of reductions in Δ_{trial} to arrive at a steplength that is sufficiently small to produce descent, it is counterproductive to follow this with an immediate doubling of Δ_{trial}. However, if the same search direction produces at least two successful iterates in a row, then this would indicate that the size of the step we are taking is probably too short, so we double Δ_{trial} in an effort to accelerate the search along that direction. If neither of the above two situations holds, then we do not alter Δ_{trial} (i.e., we choose $\ell = 0$).

On the other hand, if there is no decrease in f, in Step 3b we reduce Δ_{trial} by a factor of one-half.

In Step 4, all processes simultaneously check for convergence, each using its own locally-stored, locally-updated copy of Δ_{trial}. We note that in a heterogeneous envi-

ronment, there exists the possibility that the processes may not have identical values for Δ_{trial} because of slight differences in both storage and arithmetic for floating-point numbers; see [2]. We address this issue in further detail in §3.3.2.

Iteration:

0. For each **new best** message in my queue:
   (a) If $f_+ < f_{\text{best}}$ for an incoming triplet $\{x_+, f_+, \Delta_+\}$, then
      i. $\{x_{\text{best}}, f_{\text{best}}, \Delta_{\text{best}}\} \leftarrow \{x_+, f_+, \Delta_+\}$, and
      ii. $\Delta_{\text{trial}} \leftarrow \Delta_{\text{best}}$.
   (b) Else, discard the triplet $\{x_+, f_+, \Delta_+\}$.
1. Compute $x_{\text{trial}} \leftarrow x_{\text{best}} + \Delta_{\text{trial}}\, d$ and evaluate $f_{\text{trial}} \equiv f(x_{\text{trial}})$.
2. Set $\{x_+, f_+, \Delta_+\} \leftarrow \{x_{\text{trial}}, f_{\text{trial}}, \Delta_{\text{trial}}\}$.
3. (a) If $f_+ < f_{\text{best}}$, then
      i. $\Delta_{\text{trial}} \leftarrow \lambda\, \Delta_{\text{trial}}$ with $\lambda \in \{2^\ell \mid \ell \in \mathbb{Z}_+\}$; $\Delta_+ \leftarrow \Delta_{\text{trial}}$;
      ii. $\{x_{\text{best}}, f_{\text{best}}, \Delta_{\text{best}}\} \leftarrow \{x_+, f_+, \Delta_+\}$; and
      iii. *broadcast* a non-blocking **new best** message with the triplet $\{x_{\text{best}}, f_{\text{best}}, \Delta_{\text{best}}\}$.
   (b) Else $\Delta_{\text{trial}} \leftarrow \frac{1}{2}\Delta_{\text{trial}}$.
4. If $\Delta_{\text{trial}} > tol$, go to Step 0. Else *broadcast* a non-blocking **single direction convergence** message with the triplet $\{x_{\text{best}}, f_{\text{best}}, \Delta_{\text{best}}\}$.
5. Wait and process each incoming message in my queue until either
   (a) enough of the processes report single direction convergence for this same point or
   (b) a better point is received.
   In case (a), exit. In case (b), go to Step 0.

FIG. 3.2. *Peer-to-peer version of APPS.*

3.2. Peer-to-Peer APPS. The peer-to-peer version of APPS, from the perspective of a single process, is given in Fig. 3.2. Note that the process' local values for x_{best}, x_+, Δ_{trial}, etc. *may not always agree with the local values on other processes.* This is in contrast to PPS, where all values except f_{trial} and x_{trial} are synchronized. While PPS relies on a global reduction operation to synchronize all critical values, APPS relies on non-blocking broadcasts to exchange information between processes. Descriptions of the individual steps of APPS follow. (The initialization for APPS is unchanged from that for PPS.) As we examine these steps, keep in mind that at each step, every process decides what to do next based only on its current *local* information.

Step 0: Checking for candidates from other processes. Before a process undertakes a new evaluation of the objective function, it considers any "new best" messages that may have arrived during the previous function evaluation. The receiving process considers each incoming triplet $\{x_+, f_+, \Delta_+\}$ as a candidate for a new best; hence the test in Step 0a. To make the procedure robust, we handle tie-breaking (i.e., the case where $f_+ = f_{\text{best}}$) in a consistent fashion, the details of which are deferred to §3.3.2.

Step 1: Evaluating the function. Step 1 is the computational workhorse of parallel pattern search and is equivalent to the same step in synchronous PPS. The one substantive difference is that in PPS, x_{best} and Δ_{trial} are identical across all processes. In APPS, these values are no longer synchronized; they depend only on the information that is currently known to the process when it constructs x_{trial}.

Step 2: Assigning the local candidate. This step does not actually require any action; it is here to emphasize that, in contrast to PPS, accepting the local trial

point as a possible candidate for the new best does not involve the other processes. Instead, input from other processes is assessed in Step 0, as it becomes available.

Step 3: Assessing the local candidate. If f_+, the function value at x_+, is better than f_{best}, then Δ_{trial} (and Δ_+) are increased (using the same strategy given in §3.1 for PPS), the process' triplet $\{x_{\text{best}}, f_{\text{best}}, \Delta_{\text{best}}\}$ is updated, and a message is broadcast to the other processes to inform them of this improvement. Otherwise, the process reduces Δ_{trial} and continues.

Step 4: Checking for convergence along my search direction. There are two possible outcomes for Step 3: either f_+ replaces f_{best} (in which case, Δ_{trial} may be increased) or f_{best} is unchanged and Δ_{trial} is reduced. If the second outcome occurs and $\Delta_{\text{trial}} \leq tol$, this signals that no improvement can be found from the current x_{best} along the search direction d that this process owns and thus we may have arrived at a stationary point [9]. The process then notifies the other processes, by broadcasting a "single direction convergence" message, that it has converged (within tolerance) along its search direction.

Step 5: Waiting for a more complete picture of the entire search. The last step in APPS is the one step where a process may wait in an idle loop. Step 5 is reached only when a process has converged along its search direction. The idle process waits until either one of two things happens: it receives enough single direction convergence messages to verify global convergence to a stationary point of the objective function, or another process produces a point with a function value that is lower than f_{best}. The details regarding what constitutes "enough" single direction convergence messages are deferred to §3.3.3, where we discuss the precise measure of "enough" and how this can be determined.

3.3. Handling Messages and Exploiting Parallelism. Now that we have discussed the essential logic of APPS, we change it slightly to better handle the message traffic and to better exploit parallelism.

There are technical considerations underlying the implementation of APPS that cause us to modify the algorithm slightly from the version presented in Fig. 3.2. In particular, in the discussion above we have referred to a set of p processes, each of which handles computation, communication, and decision-making. However, it is convenient to split the computation (i.e., the evaluation of the objective function) from the communication and decision-making. One motivation for spawning a separate process to handle each evaluation of the objective function is that as a consequence of receiving a new best point from another process, it may be desirable to terminate an evaluation at some x_{trial} in order to move to the search to the new x_{best}. A second motivation is that it should eliminate the accumulation of a large number of unprocessed messages, which can cause the message queue to overflow.

We start with a group of APPS *agent* processes that are in charge of the communication and decision-making. Each evaluation of $f(x_{\text{trial}})$ is spawned as a separate process that is subservient to a single APPS agent. The result is a set of APPS agents working in peer-to-peer mode, with each APPS agent spawning function evaluation processes as necessary. In contrast with the description of APPS given in Fig. 3.2, APPS agents now dynamically initiate actions solely in response to messages, rather than routinely cycling through a fixed set of steps. It should be noted that the APPS agents require very little processing time, relative to the amount of time devoted to evaluating $f(x_{\text{trial}})$. Essentially, each APPS agent lies dormant until the arrival of an incoming message, which then triggers some action.

The types of incoming messages that an APPS agent receives are categorized as follows: a "return" from the process it spawned for the evaluation of $f(x_{\text{trial}})$, a "new best" message from another APPS agent, a "single direction convergence" message from another APPS agent, or a "shutdown" message from another APPS agent. We now investigate in more detail an APPS agent's reaction to each type of incoming message.

3.3.1. Handling "return" messages. An APPS agent receives a "return" message when the process it spawned to evaluate $f(x_{\text{trial}})$ returns the computed value f_{trial}. In Fig. 3.3 we show an APPS agent's actions in response. In the discussions that follow, we introduce here an additional item to be associated with each point—a *convergence table* Π. The convergence table is used to detect a stationary point. It lists which of the p search directions from x have converged to within tolerance. We defer a further discussion of how this information is processed to §3.3.3, where we discuss an APPS agent's action in response to a "single direction convergence" message in more detail.

Return from Evaluation of the Objective. Receive f_{trial}.

1. Update x_{best} and/or Δ_{trial}.
   (a) If $f_{\text{trial}} < f_{\text{best}}$, then
      i. $\Delta_{\text{trial}} \leftarrow \lambda\, \Delta_{\text{trial}}$ with $\lambda \in \{2^\ell \mid \ell \in \mathbb{Z}_+\}$,
      ii. $\{x_{\text{best}}, f_{\text{best}}, \Delta_{\text{best}}, \Pi_{\text{best}}\} \leftarrow \{x_{\text{trial}}, f_{\text{trial}}, \Delta_{\text{trial}}, \Pi_{\text{trial}}\}$, and
      iii. *broadcast* a non-blocking **new best** message with the quadruple $\{x_{\text{best}}, f_{\text{best}}, \Delta_{\text{best}}, \Pi_{\text{best}}\}$.
   (b) Else if x_{best} is *not* the point used to generate x_{trial}, then $\Delta_{\text{trial}} \leftarrow \Delta_{\text{best}}$.
   (c) Else $\Delta_{\text{trial}} \leftarrow \frac{1}{2}\Delta_{\text{trial}}$.
2. Check for convergence and spawn next objective function evaluation.
   (a) If $\Delta_{\text{trial}} > tol$, then compute $x_{\text{trial}} \leftarrow x_{\text{best}} + \Delta_{\text{trial}}\, d$, initialize Π_{trial} to FALSE, and *spawn* a new process to evaluate $f(x_{\text{trial}})$.
   (b) Else update Π_{best} (to signal convergence to x_{best} along my direction d) and *broadcast* a non-blocking **single direction convergence** message with the quadruple $\{x_{\text{best}}, f_{\text{best}}, \Delta_{\text{best}}, \Pi_{\text{best}}\}$.

FIG. 3.3. *APPS agent's response to a return message.*

After receiving a return message, an APPS agent first must determine if a new best point has been identified, as shown in Step 1a. If so, the step length Δ_{trial} may be increased (using the same rule as for PPS, given in §3.1) and $\{x_{\text{trial}}, f_{\text{trial}}, \Delta_{\text{trial}}, \Pi_{\text{trial}}\}$ replaces $\{x_{\text{best}}, f_{\text{best}}, \Delta_{\text{best}}, \Pi_{\text{best}}\}$. The improvement is broadcast to all other APPS agents.

Upon first inspection, the need for Step 1b may not be clear. An APPS agent constructs x_{trial} using its current values of x_{best} and Δ_{trial} (Step 2a in Fig. 3.3). While the process spawned by an APPS agent is busy evaluating $f(x_{\text{trial}})$, there is always the chance that another APPS agent will broadcast a quadruple $\{x_+, f_+, \Delta_+, \Pi_+\}$ whose value of f_+ improves upon the resident value of f_{best}. As we shall see in §3.3.2, when an APPS agent receives such an incoming message, it replaces $\{x_{\text{best}}, f_{\text{best}}, \Delta_{\text{best}}, \Pi_{\text{best}}\}$ with $\{x_+, f_+, \Delta_+, \Pi_+\}$. Before constructing the next x_{trial}, the APPS agent ascertains if, while $f(x_{\text{trial}})$ was being computed, x_{best} was replaced as the result of an incoming message from another APPS agent. If so, then this new x_{best} will be used to compute the next x_{trial}. In this case, the APPS agent replaces Δ_{trial} with Δ_{best}; using the value of Δ associated with the best point retains some scaling information.

In Step 1c, we halve Δ_{trial} after confirming that f_{best} has not been replaced either

by f_{trial} or by some f_+ contained in a message that arrived from another APPS agent while $f(x_{\text{trial}})$ was being computed.

Step 2 in Fig. 3.3 checks the value of Δ_{trial}, our measure of progress toward a solution, and reacts appropriately. If Δ_{trial} is greater than *tol* we continue the search. Otherwise, when $\Delta_{\text{trial}} \leq tol$, the search along the APPS agent's direction d has converged to x_{best}, and this information needs to be broadcast to all APPS agents.

3.3.2. Handling "new best" messages. When an APPS agent receives a "new best" message from another APPS agent, the first thing to check is whether or not the incoming quadruple really does contain the best function value seen thus far.

Here we encounter an important caveat of heterogeneous computing [2]. The comparison of floating point values (in particular, f's and Δ's) controls the flow of APPS and we depend on these comparisons to give consistent results across all processes. Therefore, we must ensure that values are only compared to a level of precision available on all processors. In other words, a "safe" comparison declares a equivalent to b if

$$\frac{|\, a - b \,|}{\max\{|a|, |b|\}} < \epsilon^*_{\text{mach}}, \tag{3.1}$$

where ϵ^*_{mach} is greater than or equal to the maximum value of machine epsilon across the values for machine epsilon on all processors participating in the computation. If both $|a|$ and $|b|$ are below ϵ^*_{mach}, then they are automatically considered equal and (3.1) is not evaluated.

The second concern raised by the concurrency of the processes is what to do when f_+ and f_{best} are equivalent. Currently, APPS uses the following tie-breaking scheme. If f_+ and f_{best} satisfy (3.1), then compare Δ_+ and Δ_{best} and select the candidate with the larger value of Δ. If Δ_+ and Δ_{best} also satisfy (3.1), check next to see if x_+ and x_{best} are the same. Rather than comparing x_+ and x_{best} directly, by computing some norm of the difference, we use a unique global identifier with which APPS tags each point. Thus, two points are considered equivalent if and only if their f-values, Δ-values, and unique global identifiers are equivalent. This means that two points that actually are equal, but were generated via different paths on different processes, will be considered to be "different" points since their global identifiers do not match. However, since the purpose of the identification is to break ties in a consistent fashion, all we need worry about is what to do when both the f-values and the Δ-values are equivalent but the global identifier is not. In this last case, ties are broken in favor of the point with the lower global identifier. Since the global identifier of each point is a unique integer, the resolution is unambiguous. So, whenever we compare f_+ and f_{best}, the comparison incorporates this tie-breaking strategy.

Now that we can assess "improvement" on f_{best} in a way that both handles the vagaries of floating-point representation and breaks ties in a consistent fashion, we examine in more detail an APPS agent's actions to a "new best" message, shown in Fig. 3.4.

Assuming improvement on f_{best}, the first action taken by an APPS agent is to determine the status of the search along the direction d. There are three possibilities to consider.

The first possibility, shown in Step 1a, is that at some point the search along d had converged within tolerance and so the APPS agent is now waiting for incoming messages to either confirm overall convergence of the search or, as in this case, to

New Best. Receive $\{x_+, f_+, \Delta_+, \Pi_+\}$.

1. If $f_+ < f_{\text{best}}$ then
   (a) If I had converged to x_{best} along my search direction d then
      i. $\{x_{\text{best}}, f_{\text{best}}, \Delta_{\text{best}}, \Pi_{\text{best}}\} \leftarrow \{x_+, f_+, \Delta_+, \Pi_+\}$, $\Delta_{\text{trial}} \leftarrow \Delta_{\text{best}}$,
      ii. compute $x_{\text{trial}} \leftarrow x_{\text{best}} + \Delta_{\text{trial}}\, d$, initialize Π_{trial} to all FALSE, and *spawn* a new process to evaluate $f(x_{\text{trial}})$.
   (b) Else if $\Delta_{\text{trial}} < \Delta_+$, then
      i. *terminate* the process evaluating $f(x_{\text{trial}})$,
      ii. $\{x_{\text{best}}, f_{\text{best}}, \Delta_{\text{best}}, \Pi_{\text{best}}\} \leftarrow \{x_+, f_+, \Delta_+, \Pi_+\}$, $\Delta_{\text{trial}} \leftarrow \Delta_{\text{best}}$,
      iii. compute $x_{\text{trial}} \leftarrow x_{\text{best}} + \Delta_{\text{trial}}\, d$, initialize Π_{trial} to all FALSE, and *spawn* a new process to evaluate $f(x_{\text{trial}})$.
   (c) Else, $\{x_{\text{best}}, f_{\text{best}}, \Delta_{\text{best}}, \Pi_{\text{best}}\} \leftarrow \{x_+, f_+, \Delta_+, \Pi_+\}$.
2. Else discard $\{x_+, f_+, \Delta_+, \Pi_+\}$.

FIG. 3.4. *APPS agent's response to a new best message.*

produce a new best point (see Step 5 in Fig. 3.2). When the latter occurs, the incoming quadruple is accepted and the search is resumed from the new x_{best}.

The second possibility, shown in Step 1b, is that the search along d is still in progress, but that the steps along d have become small, i.e., $\Delta_{\text{trial}} < \Delta_{\text{best}}$. If so, then the search along d has reduced Δ_{trial}—perhaps repeatedly—in an effort to find improvement on f_{best}. In this case, it is particularly useful to have an APPS agent acting independently of the function evaluation process. An APPS agent can terminate the current evaluation of $f(x_{\text{trial}})$ before it actually finishes (Step 1bi) in favor of starting a new evaluation of the objective based on a new value of x_{best} (Step 1biii). The question to ask is why we would choose to do so.

In certain cases, the current evaluation of the objective function is terminated in favor of starting one based on a new best point. Imagine the following scenario. Suppose three APPS agents, A, B, and C start off with the same value for x_{best}, generate their own x_{trial}'s, and spawn their own evaluations of $f(x_{\text{trial}})$. Each evaluation of the objective function takes several hours. The evaluation for Agent A completes first and there is no improvement, so Agent A reduces its step length, generates a new trial point, and spawns a new evaluation of the objective function. A few minutes later, Agent B's evaluation finishes and it produces improvement. Agent B broadcasts a "new best" message to the other APPS agents. Agent A receives this message and terminates its current evaluation of the objective function in order to move to the better point. This may save several hours of wasted computing time. However, Agent C, which is still working on its first evaluation of the objective function, waits for that to complete before considering a move to the new x_{best} because the inequality on Δ_{trial} does not hold in Step 1b of Fig. 3.4.

The third possibility when the incoming value of f_+ improves upon the local value of f_{best} is to simply accept the incoming quadruple, as shown in Step 1c. This is exactly the strategy for Agent C outlined in the scenario described above.

The final observation to be made is that if f_+ does not improve upon f_{best}, the quadruple $\{x_+, f_+, \Delta_+, \Pi_+\}$ is simply discarded; it already has been superseded by another point and thus is of no interest.

3.3.3. Handling "single direction convergence" messages. Detecting convergence for APPS is a trickier issue than it is for PPS because the APPS agents do not perform a synchronized test for convergence. Instead, each APPS agent stops spawning processes to evaluate $f(x_{\text{trial}})$ when its local value of Δ_{trial} satisfies $\Delta_{\text{trial}} \leq tol$.

Any APPS agent that arrives at this conclusion then waits until either enough other APPS agents stop at the same best point (we describe "enough" below) or until another APPS agent produces a better point from which to resume the search. Since every quadruple $\{x_{\text{trial}}, f_{\text{trial}}, \Delta_{\text{trial}}, \Pi_{\text{trial}}\}$ which improves upon f_{best} is broadcast to all APPS agents, every APPS agent eventually agrees on the best point.

Single Direction Convergence. Receive $\{x_+, f_+, \Delta_+, \Pi_+\}$.

1. If $f_+ < f_{\text{best}}$ then go through the steps for a **new best** message.
2. If $f_+ = f_{\text{best}}$ then
   (a) Update Π_{best} to include any new information contained in Π_+.
   (b) If *I* am the *temporary master*, then check for convergence.
   If enough of the other processes have converged (i.e., their associated directions form a positive spanning set), then
      i. report $\{x_{\text{best}}, f_{\text{best}}\}$,
      ii. *broadcast* a non-blocking **shutdown** message to the remaining APPS agents, and
      iii. exit.
3. Else discard $\{x_+, f_+, \Delta_+, \Pi_+\}$.

FIG. 3.5. *APPS agent's response to a single direction convergence message.*

When an APPS agent receives a "single direction convergence" message, it checks to make sure that this function value and associated point have been seen before. If not (a distinct possibility since messages may arrive out of order), then the APPS agent handles the incoming quadruple as if it were part of a "new best" message.

If the incoming point is the same as the best point we have, i.e., $f_+ = f_{\text{best}}$, then the APPS agent receiving the message must update its convergence table Π_{best} to include any new information regarding the convergence of other search directions to the same point x_{best}. Again, timing issues must be taken into account as either the sending or the receiving APPS agent may have information that has not yet been seen by the other.

Next, in order to check for convergence of a sufficient number of the p independent search directions, it is useful to have a *temporary master* to avoid redundant computation. We define the temporary master to be the APPS agent with the lowest process identification number. While this is usually process 0, it is not necessarily the case if a fault occurs; we discuss this scenario further in §4. The temporary master checks to see if the set of directions along which the search has converged forms a positive spanning set. If so, it reports to the user the final result of the search, broadcasts a "shutdown" message, and exits.

Checking for a positive spanning set can be done as follows. We know that a positive spanning set for $\mathbb{R}^n$ must contain at least $n+1$ vectors [6]. So if the convergence table has at least $n+1$ entries, it is time for the temporary master to check for convergence of the overall process. (Every APPS agent knows $\mathcal{D}$, which is why it is possible for any APPS agent to serve as temporary master.) Let $\mathcal{V} \subseteq \mathcal{D}$ be the candidate for a positive spanning set. We solve $n+1$ nonnegative least squares problems according to the following theorem.

THEOREM 3.1. *A set $\mathcal{V} = \{v_1, v_2, \ldots, v_m\}$ is a positive spanning set if the set $\mathcal{E} = \{e_1, e_2, \ldots, e_n, -\mathbf{1}\}$ is in its positive span (where $-\mathbf{1}$ is the vector of all -1's).*

Alternatively, we can check the positive basis by first verifying that $\mathcal{V}$ is a spanning set using, say, a QR factorization with pivoting, and then solving a linear program.

THEOREM 3.2 (Wright [27]). *A spanning set $\mathcal{V} = \{v_1, v_2, \ldots, v_m\}$ is a positive*

spanning set if the maximum of the following linear program is 1.

$$\max t, \text{ s.t. } Vx = 0,\ x_i \geq t\ \forall\, i,\ 0 \leq t \leq 1,$$

where V is a matrix representing the spanning set $\mathcal{V}$.

We make use of Theorem 3.1 since Netlib provides freely available software, due to Lawson and Hanson [14], for solving nonnegative least squares problems. To make use of Theorem 3.2 requires software both for QR factorizations and for the solution of linear programs; the latter is particularly difficult to come by in a freely available, portable, and easy-to-use format.

3.3.4. Handling "shutdown" messages. The reactions of the other APPS agents to a "shutdown" message from the temporary master should be clear after the discussion in §3.3.3; they are given in Fig. 3.6. Again we note the value of having both an APPS agent and a separate process for evaluating $f(x_{\text{trial}})$; once the shutdown message has been received, an APPS agent can immediately terminate the process evaluating $f(x_{\text{trial}})$ and exit.

Shutdown. Receive the shutdown message from the temporary master.
1. *Terminate* the process evaluating $f(x_{\text{trial}})$ and
2. exit.

FIG. 3.6. *APPS agent's response to a shutdown message.*

4. Fault Tolerance in APPS. The move toward a variety of computing environments, including heterogeneous distributed computing platforms, brings with it increased attention to the fault tolerance of parallel algorithms. The large size, diversity of components, and complex architecture of such systems create numerous opportunities for hardware failures, and our computational experience confirms that these failures do, in fact, occur.

In addition, the size and complexity of current simulation codes call into question the robustness of the function evaluations. For example, our experience has been that it is possible to generate input parameters that are both physically and mathematically feasible but for which the simulation codes fail to finish successfully. Thus, we must contend with software failures as well as hardware failures.

A great deal of work has been done in the computer science community with regard to fault tolerance; however, much of that work has focused on making fault tolerance as transparent to the user as possible. This often entails strategies such as checkpointing the entire state of an application to disk or replicating processes. Fault tolerance has traditionally been used with loosely-coupled distributed applications that do not depend on each other to finish, such as business database applications. This lack of interdependence is atypical of most scientific applications. While checkpointing and replication are adequate techniques for scientific applications, they incur a substantial amount of unwanted overhead; however, certain scientific applications have characteristics that can be exploited to derive more efficient and elegant stratagems for fault tolerance. Algorithm-dependent strategies for incorporating fault tolerance have already received attention in the scientific computing community; see, e.g., [21]. These approaches rely primarily on the use of diskless checkpointing, a significant improvement over traditional approaches. The nature of APPS is such

that we can even further reduce the overhead for fault tolerance and dispense with checkpointing altogether.

Two important observations should be made regarding fault tolerance in APPS. First, there are no single points of failure in the APPS algorithm itself. Assuming initialization is successful, there is just one scenario that requires a single APPS agent to coordinate efforts among all agents (i.e., the *temporary master* used to check convergence of the entire search, as shown in Fig. 3.5). However, the choice of master is not fixed. If the APPS agent serving as temporary master should fail while performing its tasks, another APPS agent steps up to take over. This means the degree of fault tolerance in APPS is constrained only by the underlying communication architecture. The current implementation of APPS uses PVM [11], which provides a rich library of communication and process management procedures needed by the APPS agents. The one limitation we inherit from PVM is that it executes multiple processes on multiple processors under the control of a single master PVM daemon. Thus the PVM daemon introduces a single point of failure within our current implementation of APPS. We expect HARNESS [1], the successor to PVM, to eliminate this disadvantage. The second observation to be made is that no checkpointing or replication of processes is necessary. The APPS agents can be reconfigured dynamically. New APPS agents require only a small packet of information from any active APPS agent in order to take over where a failed APPS agent left off. Therefore, we have been able to take advantage of algorithmic characteristics of pattern search in order to incorporate a high degree of fault tolerance into APPS with almost no additional overhead.

Having made these two observations, we now describe how fault tolerance is addressed in APPS. Every APPS agent keeps a record of active and inactive APPS agents (one per search direction), the available hosts, and a mapping of the active APPS agents to the available hosts. There are three types of faults with which we are concerned: 1) the failure of a process evaluating the objective function, 2) the failure of an APPS agent, and 3) the failure of a host processor. Once again we note the advantage of maintaining pairs of processes: an APPS agent to handle all communication (including information from PVM regarding the failure of processes) that is separate from the processes tasked with the major computations, the evaluations of the objective function. An individual APPS agent uses its record of active and inactive APPS agents to decide whether or not it is the temporary master and to determine the other APPS agents with whom it should interact in response to a failure. The responses to these three scenarios are shown in Fig. 4.1.

When a process evaluating $f(x_{\text{trial}})$ fails, the failure is reported to its master (i.e., the APPS agent that originally spawned it), and that APPS agent respawns the evaluation of the objective function at the current trial point. If several (e.g., five) attempts to evaluate the objective function fail at the same trial point, the APPS agent that was spawning those evaluations exits, triggering an APPS agent failure message to be sent to the other APPS agents. The failure of an evaluation could be handled in different ways for different applications; for instance, attempts to evaluate the objective function at a certain point could be abandoned without necessarily terminating the APPS agent.

When an APPS agent fails, all the remaining APPS agents record this failure. If the APPS agent that failed happened to be serving the role of temporary master, then another APPS agent must assume this responsibility. We maintain the convention that the active APPS agent with the lowest process number serves as temporary master. Once the question of who is temporary master is resolved, the first thing the new

- **An APPS Agent Detects Failure of its Function Evaluation Process.**
  1. If the number of attempts to evaluate $f(x_{\text{trial}})$ is less than the maximum number allowed, then respawn a new process to evaluate $f(x_{\text{trial}})$.
  2. Else exit.
- **An APPS Agent Detects Failure of Another APPS Agent.**
  1. Record the failure.
  2. If "my" process number is the lowest among the APPS agents still active, then assume the responsibility of *temporary master*.
  3. If I am the *temporary master*, then
     (a) Check for convergence. If enough of the other APPS agents report convergence (i.e., their associated directions form a positive spanning set), then
        i. report $\{x_{\text{best}}, f_{\text{best}}\}$,
        ii. *broadcast* a non-blocking **shutdown** message to the remaining APPS agents, and
        iii. exit.
     (b) If the directions corresponding to the remaining APPS agents do not form a positive spanning set, respawn all failed APPS agents on available host processors.
- **An APPS Agent Detects Failure of a Host Processor.**
  1. Remove failed host from list of available host processors.
  2. Determine all APPS agents residing on the failed host processor and treat each as a failed APPS agent.

FIG. 4.1. *Fault tolerance messages and actions.*

temporary master does is check for convergence since the now defunct APPS agent may have been in the midst of that check when it failed. If the search has not yet converged, the temporary master checks whether or not the set of directions owned by the remaining active APPS agents forms a positive spanning set. If so, then it is still possible to reliably determine whether or not the algorithm has converged, so nothing is done. Otherwise, all defunct APPS agents are restarted on the available hosts by the temporary master. Note that multiple APPS agents may be assigned to a single host.

If a host fails, the defunct host processor is removed from the list of viable hosts. The APPS agents that were running on the defunct host are regarded individually as failed APPS agents, which are then handled using the rules stated for APPS agent failures.

Despite the growing attention to fault tolerance in the parallel computing world, we are aware of only one other parallel optimization algorithm that incorporates fault tolerance, FATCOP [3]. FATCOP is a parallel mixed integer program solver that has been implemented using a Condor-PVM hybrid as the communication substrate. FATCOP is implemented in a master-slave fashion, which means that there is a single point of failure at the master process. This is addressed by having the master checkpoint information to disk (via Condor), but recovery requires user intervention to restart the program in the event of a failure. In contrast, once APPS has finished initialization, it can recover from the failure of any process of its own creation, including the failure of the temporary master. It does so on its own, with no checkpointing whatsoever.

5. Numerical Results. We compare APPS and PPS on several test problems as well as two engineering problems: a thermal design problem and a circuit simulation problem.

The tests were performed on the CPlant supercomputer at Sandia National Labs in Livermore, California. CPlant is a cluster of DEC Alpha Miata 433 MHz Processors. For our tests, we used 50 processors dedicated to our sole use.

5.1. Standard Test Problems. We compare APPS and PPS with 8, 16, 24, and 32 processors on six four-dimensional test problems [20, 5], shown in Table 5.1.

1	2	3	4	5	6
`broyden2a`	`broyden2b`	`chebyquad`	`epowell`	`toint_trig`	`vardim`

TABLE 5.1
Six standard test problems.

Since the function evaluations are extremely fast, we added extra "busy work," (in the form of solving a 100×101 nonnegative least squares problem) in order to slow down the processes evaluating f and better simulate the computational behavior of the optimization problems in which we are interested.

The parameters for APPS and PPS were set as follows. Let $n = 4$ be the problem dimension, and let $p \in \{8, 16, 24, 32\}$ be the number of processors. The first $2n$ search directions in $\mathcal{D}$ are $\{e_1, e_2, \ldots, e_n, -e_1, -e_2, \ldots, -e_n\}$. The remaining $p-2n$ directions are generated randomly (with a different seed for every run) and normalized to unit length. This construction ensures that $\mathcal{D}$ is a positive spanning set. We initialize $\Delta = 1.0$ and $tol = 0.001$. We start each of these six problems from the standard starting point [20, 5].

Method	**Process ID**	**Function Evals**	**Function Breaks**	**Init Time**	**Idle Time**	**Total Time**
APPS	0	237	66	0.17	0.00	24.72
	1	266	70	0.02	0.12	22.36
	2	302	89	0.02	0.12	24.32
	3	274	77	0.02	0.15	22.31
	4	270	62	0.02	0.04	24.56
	5	282	81	0.02	0.04	24.58
	6	273	59	0.02	0.04	24.59
	7	276	61	0.02	0.03	24.55
Summary Statistics		*272.5*	*70.6*	*0.04*	*0.07*	*24.72*
PPS	0	235	0	0.74	2.55	30.63
	1	235	0	0.39	7.23	30.28
	2	235	0	0.25	6.74	30.14
	3	235	0	0.13	6.94	30.01
	4	235	0	0.10	6.36	29.98
	5	235	0	0.07	6.51	29.95
	6	235	0	0.04	6.23	29.92
	7	235	0	0.02	6.26	29.90
Summary Statistics		*235*	*N/A*	*0.22*	*6.10*	*30.63*

TABLE 5.2
Detailed results for `epowell` *on eight processors.*

Before considering the summary results, we discuss the details of two sample runs (one each for APPS and PPS) given in Table 5.2. Each process reports its own counts and timings. All times are reported in seconds and are wall clock times. Because APPS is asynchronous, the number of function evaluations spawned by each APPS agent varies considerably. Furthermore, the APPS agents sometimes terminate ("break") processes evaluating $f(x_{\text{trial}})$. On the other hand, because PPS is synchronous, every process executes the same number of function evaluations and

there are no breaks. For both APPS and PPS, the initialization time is longer for Process 0 since it is in charge of spawning all remaining tasks. The idle time varies from process to process, but is overall lower for APPS than PPS. An APPS agent is idle only when it has converged along its search direction, but a PPS process may potentially have some idle time every iteration while it waits for the completion of the global reduction. The total wall clock time varies from process to process since each starts and stops at slightly different times. The summary information reports the mean over all eight processes, except in the case of total time, where the *maximum* total time over all eight processes is reported.

Because some of the search directions are generated randomly, every run of APPS and PPS follows a different path to the solution and generates possibly different solutions in the case of multiple minima. (The exception is PPS with $p = 8$. Because there are no "extra" search directions, the path to the solution is the same for every run—only the timings differ. The nondeterministic nature of APPS causes us to see different counts and different timings for every run, even if the search directions for each run are identical.) Therefore, for each problem in Table 5.1 we report the mean of the summary statistics from 25 runs; for each *individual* run we collected the same summary statistics (except the initialization time) reported in Table 5.2.

Prob No.	**No. Procs**	**Function Evals**		**APPS Breaks**	**Idle Time**		**Total Time**	
		APPS	**PPS**		**APPS**	**PPS**	**APPS**	**PPS**
1	8	40.59	37.00	8.14	0.07	0.95	3.88	4.88
	16	41.77	40.12	7.93	0.02	2.04	3.98	6.68
	24	38.30	37.36	6.98	0.02	4.68	3.80	9.33
	32	36.57	37.92	6.88	0.03	7.81	3.83	12.81
2	8	40.35	37.00	8.28	0.06	0.97	3.84	4.92
	16	41.07	39.11	7.38	0.02	2.06	3.95	6.62
	24	38.47	39.60	7.20	0.02	4.77	3.77	9.68
	32	35.10	36.76	6.23	0.03	7.04	3.72	11.92
3	8	73.06	62.00	16.74	0.05	1.61	6.86	8.11
	16	48.33	40.44	9.54	0.02	2.11	4.69	6.92
	24	45.67	38.64	9.26	0.02	4.59	4.47	9.39
	32	44.34	37.60	9.14	0.04	7.54	4.59	12.56
4	8	272.29	235.00	68.27	0.30	6.64	24.50	30.48
	16	139.63	153.04	37.39	0.05	8.04	12.24	24.76
	24	139.38	126.96	36.40	0.03	14.10	12.26	28.46
	32	98.88	102.64	26.20	0.03	28.07	9.41	41.03
5	8	53.83	41.00	10.97	0.04	1.11	4.99	5.60
	16	51.40	39.12	10.47	0.02	1.97	4.91	6.51
	24	47.86	36.88	9.24	0.02	4.43	4.69	9.03
	32	45.90	33.04	8.70	0.04	6.41	4.81	10.83
6	8	205.39	77.00	51.24	0.05	2.00	18.15	9.97
	16	101.46	80.44	25.58	0.02	3.97	8.93	12.83
	24	72.44	49.96	17.19	0.02	5.61	6.57	11.63
	32	64.09	46.04	15.96	0.03	9.58	6.14	15.51

TABLE 5.3

Summary statistics (across 25 runs) for the four-dimensional test problems shown in Table 5.1.

The test results are summarized in Table 5.3. These tests were executed in what should have been a particularly favorable environment for PPS—a cluster of homogeneous, dedicated processors. The primary difficulty for PPS is the cost of synchronization in the global reduction. In terms of average function evaluations per processor, APPS and PPS typically required about the same number. In general, for both APPS and PPS, the number of function evaluations per processor decreased as the number of processes increased. We expected the idle time for APPS to be less than that for

PPS; and, indeed, the idle time is two orders of magnitude less. Furthermore, the idle time for PPS increases as the number of processors goes up. APPS was faster (on average) than PPS in 23 out of 24 cases. The total time (on average) for APPS either stayed more or less steady or actually decreased as the number of processors increased. In contrast, the total time (on average) for PPS almost always increased as the number of processors increased, due to the synchronization penalty incurred with the addition of more processes.

Comparing APPS and PPS on simple problems is not necessarily indicative of results for typical engineering problems. The results in §5.2 and §5.3 yield more meaningful comparisons, given the types of problems for which pattern search is best suited.

5.2. TWAFER: A Thermal Design Problem. In this set of tests, the engineering application is an optimal control problem for a thermal deposition furnace for silicon wafers. The furnace contains a vertical stack of wafers and several heater zones. The goal is to choose power settings for the heaters in each of n zones to achieve a prescribed constant temperature across each wafer and throughout the stack. The simulation code, TWAFER [12], yields measurements at a discrete collection of points on the wafers. The objective function f is defined as

$$f(x) = \sum_{j=1}^{N} \left(T_j(x) - T_*\right)^2 , \tag{5.1}$$

where N is the number of discrete temperature measurement points, $T_j(x)$ is the simulated temperature at the jth point for the power settings defined by x, and T_* is the prescribed ideal temperature.

We consider the four- and seven-zone (or variable) problems with $N = 40$ and $N = 400$, respectively. For the four-zone problem, the initial guess produced a function value of 2.26×10^6. The initial guess for the seven zone problem produced a function value of 7.43×10^4. (The initial guess for the seven-zone problem was much closer to the final solution.)

We used the following settings for APPS and PPS. The first $n+1$ search directions are the points of a regular simplex centered about the origin. The remaining $p-n-1$ points are generated randomly and normalized to unit length. Because the magnitude of the variables was $O(100)$, we set $\Delta = 10.0$. Note that it can be quite useful to choose the initial Δ based on the magnitudes of the components in x_0 as a way to capture some scaling information about the problem [25]. We chose $tol = 0.1$, which corresponds to a level of accuracy that is reasonable in the power settings.

There are some difficulties from the implementation point of view that are quite common when dealing with simulation codes. Because TWAFER is a legacy code, it expects an input file with a specific name and produces an output file with a specific name. The names of these files cannot be changed, and TWAFER cannot be hooked directly to PVM. As a consequence, we must write a "wrapper" program that runs an input filter, executes TWAFER via a system call, and runs an output filter. Because TWAFER is executed via a system call, APPS has no way of terminating its execution prematurely. (APPS can terminate the wrapper program, but TWAFER itself will continue to run, consuming system resources.) Therefore, we allow all function evaluations to run to completion; that is, we do not allow any breaks.

Another feature of TWAFER is that there are nonnegativity constraints on the power settings. The solution is known to be strictly positive, and the constraints

play only a minor role in finding the solution. We did not invoke TWAFER at any point that had one or more negative components; to accomplish this, we use a simple barrier function that returns a large value (e.g., 10^{50}). This is a classic trick used by direct search methods for dealing with bound constraints. With the correct choice of $\mathcal{D}$, pattern search methods that use such a strategy can be shown to have at least one subsequence of iterates that converge to a Karush-Kuhn-Tucker point [18, 19].

Problem	**Method**	**Procs**	**f(x∗)**	**Function Evals**	**Idle Time**	**Total Time**
4 Zone	APPS	20	0.67	334.6	0.17	395.94
4 Zone	PPS	20	0.66	379.9	44.77	503.88
7 Zone	APPS	35	3.30	240.4	71.48	2260.46
7 Zone	PPS	35	2.85	202.2	213.90	2306.83

TABLE 5.4
Summary statistics (across multiple runs) for the four- and seven-zone TWAFER problems.

Results for the TWAFER problem are given in Table 5.4. The four-zone results report the means across all twenty processors over all ten runs. The seven-zone results report the means across all 35 processors over all nine runs. (We started ten runs for the seven-zone problem. One of the ten PPS runs failed due to a processor fault. One of the ten APPS runs experienced several faults and, although it did get the final solution, the summary data was incomplete.)

Recall that the goal is to choose power settings to achieve a constant temperature across each wafer and throughout the stack. In Fig. 5.1 we show the temperatures computed by TWAFER at each wafer along a line of discretization points from the bottom to the top of the furnace. We show results for both the initial settings we were given for the seven-zone problem and the best and worst settings returned by APPS, corresponding to function values of 1.48 and 7.74, respectively. (The plots of the results from the best and worst PPS solutions are indistinguishable from the best and worst plots for APPS.) Table 5.4 shows that for this problem, on average, PPS yields slightly better function values than APPS (less than 1/1000th of a percent relative difference compared to the function value at the starting point) but required more total time. Fig. 5.1 demonstrates that qualitatively, all the solutions produced were comparable, particularly given the modest choice of $tol = 0.1$.

Clearly, the idle time figures prominently in the overall performance of PPS. The average simulation time is 1.3 seconds for the four-zone problems and 10.4 seconds for the seven-zone problem. However, when the nonnegativity constraints are violated, TWAFER is not called, so the execution time is essentially zero since we simply return 10^{50} after checking the coordinates of x_{trial}. The relatively high mean idle time for APPS (for the seven-zone problem) can be traced to a single run for which the idle time was particularly high for some processors (634 seconds on average across all 35 processors); on the remaining runs, the average APPS idle time per processor was lower by several orders of magnitude. We were unable to determine the cause of the unusually large idle time.

5.3. SPICE: A Circuit Simulation Problem. The problem is to match simulation data to experimental data for a particular circuit in order to determine its characteristics. In our case, we have 17 variables representing inductances, capacitances, diode saturation currents, transistor gains, leakage inductances, and transformer core

FIG. 5.1. *TWAFER results for the seven-zone problem using APPS with tol = 0.1. The solid line represents the simulation output for the best settings found by APPS and the dotted line represents the simulation output for the worst settings found by APPS. The dashed line represents the simulation output for the initial settings. The target is a constant temperature of 1300.*

parameters. The objective function is defined as

$$f(x) = \sum_{j=1}^{N} \left(V_j^{\mathrm{SIM}}(x) - V_j^{\mathrm{EXP}} \right)^2 , \tag{5.2}$$

where N is the number of time steps, $V_j^{\mathrm{SIM}}(x)$ is the simulation voltage at time step j for input x, and V_j^{EXP} is the experimental voltage at time step j.

The SPICE3 [22] package is used for the simulation. Like TWAFER, SPICE3 communicates via file input and output and so we again use a wrapper program.

The input filter for SPICE is more complicated than that for TWAFER because the variables for the problem are on different scales. Since APPS has no mechanism for scaling, we handled this within the input filter by computing an affine transformation of the variables used to formulate the objective function (5.2). Additionally, all the variables have upper and lower bounds. Once again, we use a simple barrier function.

The output filter for SPICE is also more complicated than that for TWAFER. The SPICE output files consist of voltages that are to be matched to the experimental data. The experimental data is two cycles of output voltage measured at approximately $N = 2700$ discrete time steps (see Fig. 5.2). The simulation data contains approximately 10 or more cycles, but only the last few complete cycles are used because the early cycles are not stable. The cycles must be automatically identified so that the data can be

aligned with the experimental data. Furthermore, the time steps from the simulation may differ from the time steps in the experiment, and so the simulation data is interpolated (piecewise constant) to match the experimental data. The function value at the initial point is 465.

FIG. 5.2. *Spice results using APPS with tol* $= 0.1$. *The solid line represents the experimental output. The dashed line represents the simulation output after optimization. The dotted line represents the simulation output for the initial point.*

The APPS parameters were set as follows. The search directions were generated in the same way as those for the test problems in §5.1. We set $\Delta = 1.0$ (the affine transformation means the variables are well-scaled) and *tol* is 0.1 (the tolerance corresponds to a less than 1% change in the circuit parameter). Once again, we do not allow "breaks" since the function evaluation is called from a wrapper program via a system call.

Method	**Procs**	$\mathbf{f(x*)}$	**Function Evals**	**Idle Time**	**Total Time**
APPS	34	26.3	57.5	111.92	1330.55
APPS	50	26.9	50.6	63.22	807.29
PPS	34	28.8	53.0	521.48	1712.24
PPS	50	34.9	47.0	905.48	1646.53

TABLE 5.5
Results (one run each) for the 17 variable SPICE problem.

The results from APPS and PPS on the SPICE problem are reported in Table 5.5.

In this case, we are reporting the results of single runs; we give results for 34 and 50 processors. The average SPICE run time is approximately 20 seconds; however, once again we do not differentiate between times when the boundary conditions are violated and when the SPICE code is actually executed. Increasing the number of processors by 47% results in a 39% reduction in execution time for APPS but only 4% for PPS. For both 34 and 50 processors, APPS is faster than PPS, and even produces a slightly better objective value (compared to the starting value of more than 400). At the solution, two constraints are binding.

Initial Procs	**Final Procs**	**f(x∗)**	**Total Time**
34	34	27.8	1618.46
50	32	54.2	1041.14

TABLE 5.6

APPS results for the 17 variable SPICE with a failure approximately every 30 seconds.

Table 5.6 shows the results of running APPS with faults. In this case, we used a program that automatically killed one PVM process every 30 seconds. The PVM processes are the APPS agents and the wrapper programs. The SPICE3 simulation is executed via a system call, and so continues to execute even if its wrapper terminates; regardless, the SPICE3 program can no longer communicate with APPS and is effectively dead.

The results are quite good. In the case of 34 processors, every APPS task that fails must be restarted in order to maintain a positive basis. So, the final number of APPS processes is 34. The total time is only increased by 21% despite approximately 50 failures; furthermore, this time is still faster than PPS. In the case of 50 processors, the final number of processors is 32. (Recall that tasks are only restarted if there are not enough remaining to form a positive basis.) In the case of 50 processors, the solution time is only increased by 29% with faults, and is once again still faster than PPS. In this case, however, the quality of the solution is degraded. This is likely due to the fact that the solution lies on the boundary and some of the search directions that failed were needed to ensure convergence to a KKT point (see [18, 19]).

6. Conclusions. Our preliminary numerical results make clear that because APPS dynamically initiates actions solely in response to messages, it is a more effective method—even in a homogeneous cluster environment—than PPS, where "more effective" means that APPS requires less total time to return results that are comparable to those returned by PPS. We expect the differences to be even more pronounced for larger problems (where by "larger" we mean in terms of both the execution time and the number of variables) and for heterogeneous cluster computing environments. Unlike PPS, which routinely cycles through a fixed set of steps, APPS does not have any required synchronizations and, thus, appears to gain most of its advantage by reducing idle time.

Further, APPS is a fault-tolerant algorithm. We accomplish this by making algorithmic changes to parallel pattern search that introduce almost no additional overhead. As we saw in the results for the SPICE problem solved using 34 processors (§5.3), APPS does not suffer much slow-down when faults do occur.

Finally, in forthcoming work, Kolda and Torczon [13] will show that in the unconstrained case, APPS is globally convergent (even when faults occur) under the standard assumptions for pattern search [16, 26].

These features duly noted, we are investigating further improvements to the implementation of APPS. For instance, in the implementation described here, each APPS agent is responsible for exactly one process to evaluate the objective function. For multi-processor (MPP) compute nodes, this means there will be multiple agents per node. An alternative implementation of APPS is being developed in which there is exactly one agent per node, with the single agent managing multiple evaluations of the objective function. As part of this alternative implementation, the ability to dynamically add new hosts as they become available (or to re-add previously failed hosts) also will be incorporated.

Another improvement to the implementation will be the addition of a cache to store the values of the function at all the points visited by the search in order to avoid reevaluating the same point more than once. The challenges are to make the recovery of this information fast and to decide when two points are actually equal. The latter is especially difficult when we do not know the sensitivity of the function to changes in each variable.

The importance of positive bases in the pattern also raises several interesting questions. In general, we might consider the best way to generate the starting basis. The analysis of pattern search makes clear that the "conditioning" of the positive basis has an effect on the amount of decrease that may be realized [16]. Our numerical studies have indicated that the quality of the positive basis can, indeed, affect the progress of the search. Thus, explicitly monitoring the conditioning of the positive basis, which changes dynamically, could improve the overall performance of APPS. Further, supposing that enough failures have occurred so that there is no longer a positive basis, we may ask if we can easily determine the smallest number of vectors to add to once again have a positive basis. Our current implementation simply restarts all failed APPS agents (see Figure 4.1). In general, we desire a pattern that maximizes the probability of maintaining a well-conditioned positive basis in the event of failures, without requiring us to keep a large number of processes active when it is neither necessary nor convenient to do so.

Finally, although the engineering examples used in this work have bound constraints, the current version of APPS does not handle constraints in a rigorous fashion. The poor results on the SPICE problem with faults on 50 processors may well be attributed to this fact since several constraints are active at a known solution. The analysis for pattern search suggests several algorithmic options we could pursue [17, 18, 19], but the challenge is to do so in a way that works effectively within the asynchronous framework we have devised. Future work will explore robust extensions for handling constraints.

Acknowledgments. We thank Jim Kohl, Ken Marx, and Juan Meza for helpful comments and advice in the implementation of APPS and the test problems. We also thank the referees for their careful reading of an earlier draft of this manuscript. Their suggestions considerably improved the presentation of this work.

REFERENCES

[1] M. Beck, J. J. Dongarra, G. E. Fagg, G. A. Geist, P. Gray, J. Kohl, M. Migliardi, K. Moore, T. Moore, P. Papadopoulous, S. L. Scott, and V. Sunderam, *HARNESS: A next generation distributed virtual machine*, Future Generation Computer Systems, 15 (1999), pp. 571–582.

[2] L. S. Blackford, A. Cleary, A. Petitet, R. C. Whaley, J. Demmel, I. Dhillon, H. Ren, K. Stanley, J. Dongarra, and S. Hammarling, *Practical experience in the numerical*

dangers of heterogeneous computing, ACM Transactions on Mathematical Software, 23 (1997), pp. 133–147.

[3] Q. Chen and M. C. Ferris, *FATCOP: A fault tolerant Condor-PVM mixed integer program solver*, Mathematical Programming Tech. Rep. 99–05, Computer Sciences Department, University of Wisconsin, Madison, WI, 1999. To appear in SIAM Journal on Optimization.

[4] D. Conforti and R. Musmanno, *Convergence and numerical results for a parallel asynchronous quasi-Newton method*, Journal of Optimization Theory and Applications, 84 (1995), pp. 293–310.

[5] A. R. Conn, N. I. M. Gould, and P. L. Toint, *Testing a class of methods for solving minimization problems with simple bounds on the variables*, Mathematics of Computation, 50 (1988), pp. 399–430.

[6] C. Davis, *Theory of positive linear dependence*, American Journal of Mathematics, 76 (1954), pp. 733–746.

[7] J. E. Dennis, Jr. and V. Torczon, *Direct search methods on parallel machines*, SIAM Journal on Optimization, 1 (1991), pp. 448–474.

[8] E. D. Dolan, *Pattern search behavior in nonlinear optimization.* Honors Thesis, Department of Computer Science, College of William & Mary, Williamsburg, VA, May 1999.

[9] E. D. Dolan, R. M. Lewis, and V. J. Torczon, *On the local convergence properties of pattern search*, Tech. Rep. 2000–36, Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA, September 2000.

[10] H. Fischer and K. Ritter, *An asynchronous parallel Newton method*, Mathematical Programming, 42 (1988), pp. 363–374.

[11] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam, *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Network Parallel Computing*, MIT Press, Cambridge, Massachusetts, 1994.

[12] W. G. Houf, J. F. Grcar, and W. G. Breiland, *A model for low pressure chemical vapor deposition in a hot-wall tubular reactor*, Materials Science Engineering, B, Solid State Materials for Advanced Technology, 17 (1993), pp. 163–171.

[13] T. G. Kolda and V. Torczon, *On the convergence of asynchronous parallel direct search.* In preparation, 2001.

[14] C. L. Lawson and R. J. Hanson, *Solving Least Squares Problems*, SIAM, Philadelphia, 1995.

[15] R. M. Lewis, V. Torczon, and M. W. Trosset, *Why pattern search works*, Optima, 59 (1998), pp. 1–7.

[16] R. M. Lewis and V. J. Torczon, *Rank ordering and positive bases in pattern search algorithms*, Tech. Rep. 96–71, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, VA, 1996.

[17] ———, *A globally convergent augmented Lagrangian pattern search algorithm for optimization with general constraints and simple bounds*, Tech. Rep. 98–31, Institute for Computer Applications in Science and Engineering, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23681–2199, July 1998. In review for SIAM Journal on Optimization.

[18] ———, *Pattern search algorithms for bound constrained minimization*, SIAM Journal on Optimization, 9 (1999), pp. 1082–1099.

[19] ———, *Pattern search methods for linearly constrained minimization*, SIAM Journal on Optimization, 10 (2000), pp. 917–941.

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

[21] J. S. Plank, Y. Kim, and J. Dongarra, *Fault tolerant matrix operations for networks of workstations using diskless checkpointing*, Journal of Parallel and Distributed Computing, 43 (1997), pp. 125–138.

[22] T. Quarles, A. R. Newton, D. O. Perderson, and A. Sangiovanni-Vincentelli, *SPICE3 version 3f3 user's manual*, tech. rep., Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA, May 1993.

[23] S. Smith, E. Eskow, and R. B. Schnabel, *Adaptive, asynchronous stochastic global optimization for sequential and parallel computation*, in Large-Scale Numerical Optimization, T. F. Coleman and Y. Li, eds., SIAM, Philadelphia, 1990, pp. 207–227.

[24] T. L. Sterling, J. Salmon, D. J. Becker, and D. F. Savarese, *How to Build a Beowulf: A Guide to the Implementation and Application of PC Clusters*, MIT Press, Cambridge, Massachusetts, 1999.

[25] V. Torczon, *PDS: Direct search methods for unconstrained optimization on either sequential or parallel machines*, Tech. Rep. TR92–09, Rice University, Department of Computational and Applied Mathematics, Houston, TX, 1992.

[26] ———, *On the convergence of pattern search algorithms*, SIAM Journal on Optimization, 7

(1997), pp. 1–25.
[27] S. E. Wright, *A note on positively spanning sets*, American Mathematical Monthly, 107 (2000), pp. 364–366.