SAND2000-8213
Unlimited Release
Printed January 2000

Asynchronous Parallel Pattern Search for Nonlinear Optimization

P. D. Hough*‡, T. G. Kolda†‡
Computational Sciences and Mathematics Research Department
Sandia National Laboratories
Livermore, CA 94551-9217

V. J. Torczon§
Department of Computer Science
College of William and Mary
Williamsburg, VA 23187-8795

ABSTRACT

Parallel pattern search (PPS) can be quite useful for engineering optimization problems characterized by a small number of variables (say 10-50) and by expensive objective function evaluations such as complex simulations that take from minutes to hours to run. However, PPS, which was originally designed for execution on homogeneous and tightly-coupled parallel machine, is not well suited to the more heterogenous, loosely-coupled, and even fault-prone parallel systems available today. Specifically, PPS is hindered by synchronization penalties and cannot recover in the event of a failure. We introduce a new asynchronous and fault tolerant parallel pattern search (APPS) method and demonstrate its effectiveness on both simple test problems as well as some engineering optimization problems.

Keywords: asynchronous parallel optimization, pattern search, direct search, fault tolerance, distributed computing, cluster computing.

*Email: pdhough@ca.sandia.gov.
†Corresponding author. Email: tgkolda@ca.sandia.gov.
‡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: va@cs.wm.edu. This research was supported by the National Science Foundation under Grant No. CCR-9734044.

1 Introduction

We are interested in solving the unconstrained nonlinear optimization problem:

$$\text{minimize } f(x),$$

where $x \in \mathbf{R}^n$ and $f : \mathbf{R}^n \rightarrow \mathbf{R}$.

We introduce a family of asynchronous parallel pattern search (APPS) methods. Pattern search [15] is a class of direct search methods which admits a wide range of algorithmic possibilities. Because of the flexibility afforded by the definition of pattern search [23, 16], we can adapt it to the design of nonlinear optimization methods that are intended to be effective on a variety of parallel and distributed computing platforms.

Our motivations are several. First, the optimization problems of interest to us are typically defined by computationally expensive computer simulations of complex physical processes. Such a simulation may take anywhere from a few seconds to many hours of computation on a single processor. As we discuss further in §2, the dominant computational cost for pattern search methods lies in these objective function evaluations. Even when the objective function is inexpensive to compute, the relative cost of the additional work required within a single iteration of pattern search is negligible.

Given these considerations, one feature of pattern search we exploit is that it can compute multiple, independent function evaluations simultaneously in an effort both to accelerate the search process and to improve the quality of the result obtained. Thus, our approach can take advantage of parallel and distributed computing platforms.

We also have a practical reason, independent of the computational environment, for using pattern search methods for the problems of interest. Simply put, for problems defined by expensive computer simulations of complex physical processes, we often cannot rely on the gradient of f to conduct the search. Typically, this is because no procedure exists for the evaluation of the gradient and the creation of such a procedure has been deemed untenable. Further, approximations to the gradient may prove unreliable. For instance, if the accuracy of the function can only be trusted to a few significant decimal digits, it is difficult to construct reliable finite-difference approximations to the gradient. Finally, while the theory for pattern search assumes that f is continuously differentiable, pattern search methods can be effective on nondifferentiable (and even discontinuous) problems precisely because they do not explicitly rely on derivative information to drive the search. Thus we focus on pattern search for both practical and computational reasons.

However, both the nature of the problems of interest and the features of the current distributed computing environments raise a second issue we address in this work. The original investigation into parallel pattern search (PPS) methods[1] [7, 22] made two

[1]The original investigations focused on parallel direct search (PDS), a precursor to the more general PPS methods discussed here.

fundamental assumptions about the parallel computation environment: 1) that the processors were both homogeneous and tightly coupled and 2) that the amount of time needed to complete a single evaluation of the objective was effectively constant. It is time to reexamine these two assumptions.

Clearly, given the current variety of computing platforms including distributed systems comprising loosely-coupled, often heterogeneous, commercial off-the-shelf components [21], the first assumption is no longer valid. The second assumption is equally suspect. The standard test problems used to assess the effectiveness of a nonlinear optimization algorithm typically are closed-form, algebraic expressions of some function. Thus, the standard assumption that, for a fixed choice of n, evaluations complete in constant time is valid. However, given our interest in optimizing problems defined by the simulations of complex physical processes, which often use iterative numerical techniques themselves, the assumption that evaluations complete in constant computational time often does not hold. In fact, the behavior of the simulation for any given input is difficult to assess in advance since the behavior of the simulation can vary substantially depending on a variety of factors.

For both the problems and computing environments of interest, we can no longer assume that the computation proceeds in lockstep. A single synchronization step at the end of every iteration, such as the global reduction used in [22], is neither appropriate nor effective when any of the following factors holds: function evaluations complete 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. Again our goal is to introduce a class of 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.

The third, and final, consideration we address in this paper is incorporating fault tolerant strategies into the APPS methods since one intent is to use this software on large-scale heterogeneous systems. The combination of commodity parts and shared resources raises a growing concern about the reliability of the individual processors participating in a computation. If we embark on a lengthy computation, we want reasonable assurance of producing a final result, even if a subset of processors fail. Thus, our goal is to design methods that anticipate such failures and respond to 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 comparing APPS and PPS on both standard and engineering optimization test problems; and 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, 8], though the assumptions underlying these approaches differ markedly from those we address. Specifically, both assume that solving a linear system of equations 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 [20] considers the use of quasi-Newton methods, but in the context of developing asynchronous stochastic global optimization algorithms. For now, we focus on finding local minimizers.

2 Parallel Pattern Search

Before proceeding to a discussion of our APPS methods, let us first review the features of direct search, in general, and pattern search, in particular.

Direct search methods are characterized by neither requiring nor explicitly approximating derivative information. In the engineering literature, direct search methods are often called zero-order methods, as opposed to first-order methods (such as the method of steepest descent) or second-order methods (such as Newton's method) to indicate the highest order term being used in the local Taylor series approximation to f. This characterization of direct search is perhaps the most useful in that it emphasizes that in higher-order methods, derivatives are used to form a local approximation to the function, which is then used to derive a search direction and predict the length of the step necessary to realize decrease. Instead of working with a local approximation of f, direct search methods work directly with f.

Pattern search methods comprise a subset of direct search methods. While there are rigorous formal definitions of pattern search [16, 23], 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, standard global convergence results can be obtained.

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 pattern search algorithm. At iteration k, we have an iterate $x_k \in \mathbf{R}^n$ and a step-length parameter $\Delta_k > 0$. The pattern of p points is denoted by $\mathcal{D} = \{d_1, \ldots, d_p\}$. For the purposes of our simple example, we choose $\mathcal{D} \equiv \{e_1, \ldots, e_n, -e_1, \ldots, -e_n\}$ where e_j represents the jth unit vector. As we discuss at the end of this section, other choices for $\mathcal{D}$ are possible. We now have several algorithmic options open to us. One possibility is to look successively at the pattern points $x_k + \Delta_k d_i$, $i \in \{1, \ldots, 2n\}$ until either we find a point x_+ for which $f(x_+) < f(x_k)$ or we exhaust all $2n$ possibilities. At the other extreme, we could determine $x_+ \in \{x_k + \Delta_k d_i, i = 1, \ldots, 2n\}$ such that $f(x_+) = \min\{f(x_k + \Delta_k d_i), i = 1, \ldots, 2n\}$ (which requires us to compute $f(x_k + \Delta_k d_i)$

for all $2n$ vectors in the set D). Fig. 1 illustrates the pattern of points among which we search for x_+ when $n = 2$.

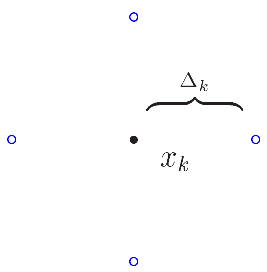

Figure 1: A simple instance of pattern search

In either variant of pattern search, if none of the pattern points reduces the objective, then we set $x_{k+1} = x_k$ and reduce Δ by setting $\Delta_{k+1} = \frac{1}{2}\Delta_k$[2]; otherwise, we set $\Delta_{k+1} = \Delta_k$ and update $x_{k+1} = x_+$. We repeat this process until some suitable stopping criterion, such as $\Delta_k < tol$, is satisfied.

There are several things to note about the two search strategies we have just outlined. First, even though we have the same pattern in both instances, we have two different algorithms with different search strategies that could conceivably produce different sequences of iterates and even different local minimums. Second, the design of the search strategies reflects some intrinsic assumptions about the nature of both the function and the computing environment in which the search is to be executed. Clearly the first strategy, which evaluates only one function value at a time, was conceived for execution on a single processor. Further it is a cautious strategy that computes function values only as needed, which suggests a frugality with respect to the number of function evaluations to be allowed. The second strategy could certainly be executed on a single processor, and one could make an argument as to why there could be algorithmic advantages in doing so, but it is also clearly a strategy that can easily make use of multiple processors. It is straightforward to then derive PPS from this second strategy, as illustrated in Fig. 2.

Before proceeding to a description of APPS, however, we need to make one more remark about the pattern. As we have already seen, we can easily derive two different search strategies using the same basic pattern. Our requirements on the outcome of the search are mild. If we fail to find a point that reduces the value of f at x_k, then we must try again with a smaller value of Δ_k. Otherwise, we accept as our new iterate any point from the pattern that produces decrease. In the latter case, we may choose to modify Δ_k. In either case, we are free to make changes to the pattern to be used in the next iteration, though we left the pattern unchanged in the examples given above. However, changes to either the step length parameter or the pattern are subject to certain algebraic conditions, outlined fully in [16].

[2]The reduction parameter is usually $\frac{1}{2}$ but can be any number in the set $(0, 1)$.

Initialization:

- Select a pattern $\mathcal{D} = \{d_1, \ldots, d_p\}$.
- Select a step-length parameter Δ_0.
- Select a stopping tolerance *tol*.
- Select a starting point x_0 and evaluate $f(x_0)$.

Iteration:

1. Evaluate $f(x_k + \Delta_k d_i)$, for $i = 1, \ldots, p$, *concurrently.*
2. Determine 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 set $x_{k+1} = x_+$ and $\Delta_{k+1} = \Delta_k$.
   Else set $x_{k+1} = x_k$ and $\Delta_{k+1} = \frac{1}{2}\Delta_k$.
4. If $\Delta_{k+1} < tol$, exit. Else, $k \leftarrow k + 1$ and repeat.

Figure 2: The PPS Algorithm

There still remains the question of what constitutes an acceptable pattern. We borrow the following technical definition from [6, 16]: a pattern must be a positive spanning set for $\mathbf{R}^n$. In addition, we add the condition that the spanning set be composed of rational vectors.

Definition 1 *A set of vectors $\{d_1, \ldots, d_p\}$ positively spans $\mathbf{R}^n$ if any vector $z \in \mathbf{R}^n$ can be written as a nonnegative linear combination of the vectors in the set; i.e., for any $z \in \mathbf{R}^n$ there exists $\alpha_1, \alpha_2, \ldots, \alpha_p \geq 0$ such that*

$$z \ = \ \alpha_1 d_1 + \cdots + \alpha_p d_p.$$

A positive spanning set contains at least $n+1$ vectors [6]. It is trivial to verify that the set of vectors $\{e_1, e_2, -e_1, -e_2\}$ (used to define the pattern for our examples above) is a positive spanning set.[3]

[3]The terminology "positive" spanning set is a misnomer; a more proper name would be "nonnegative" spanning set.

3 Asynchronous Parallel Pattern Search

Inefficiencies in processor utilization for the PPS algorithm shown in Fig. 2 arise when the objective function evaluations do not complete in approximately the same amount of time. This can 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 load on the individual processors may vary. Last, groups of processors participating in the calculation may possess different computational characteristics. When the objective function evaluations take varying amounts of time those processors that can complete their share of the computation more quickly wait for the remaining processors to contribute their results. Thus, adding more processors (and correspondingly more search directions) can actually slow down the PPS method given in Fig. 2 because of an increased synchronization penalty.

The limiting case of a slow objective function evaluation is when one never completes. This could happen if some processor fails during the course of the calculations. In that situation, the entire program would hang at the next synchronization point. Designing an algorithm that can handle failures plays some role in the discussion in this section and is given detailed coverage in the next.

The design of APPS addresses the limitations of slow and failing objective function evaluations and is based on a *peer-to-peer* approach rather than *master-slave*. Although the master-slave approach has advantages, the critical disadvantage is that, although recovery for the failure of slave processes is easy, we cannot automatically recover from failure of the master process.

In the *peer-to-peer* scenario, all processes have equal knowledge, and each process is in charge of a single direction in the search pattern $\mathcal{D}$. In order to fully understand APPS, let us first consider the single processor's algorithm for synchronous PPS in a peer-to-peer mode, as shown in Fig. 3. Here subscripts have been dropped to illustrate how the process handles the data. The set of directions from all the processes forms a positive spanning set. With the exception of initialization and finalization, the only communication a process has with its peers is in the global reduction in Step 2. To terminate, all processors detect convergence at the same time since they all have identical, albeit independent, values for Δ_{trial}.[4]

In an asynchronous peer-to-peer version of PPS (see Fig. 4), we allow each process to maintain its own versions of x_{best}, x_+, Δ_{trial}, etc. Unlike synchronous PPS, these values *may not always agree with the values on the other processes.* Each process decides what to do next based only on the current information available to it. If it finds a point along its search direction that improves upon the best point it knows so far, then it broadcasts a message to the other processors letting them know. It also checks for messages from other processors, and replaces its best point with the

[4]In a heterogeneous environment, there is some danger that the processors may not all have the same value for Δ_{trial} because of slight differences in arithmetic and the way values are stored; see [2].

Iteration:

1. Compute $x_{\text{trial}} \leftarrow x_{\text{best}} + \Delta_{\text{trial}}\, d$ and evaluate $f_{\text{trial}} = f(x_{\text{trial}})$ *(where d is "my" direction).*
2. Determine f_+ (and the associated x_+) via a *global reduction* minimizing the f_{trial} values computed in Step 1.
3. If $f_+ < f_{\text{best}}$, then $\{x_{\text{best}}, f_{\text{best}}\} \leftarrow \{x_+, f_+\}$. Else $\Delta_{\text{trial}} \leftarrow \frac{1}{2}\Delta_{\text{trial}}$.
4. If $\Delta_{\text{trial}} > tol$, go to Step 1. Else, exit.

Figure 3: Peer-to-peer version of (synchronous) PPS

Iteration:

0. Consider each incoming triplet $\{x_+, f_+, \Delta_+\}$ received from another processor. If $f_+ < f_{\text{best}}$, then $\{x_{\text{best}}, f_{\text{best}}, \Delta_{\text{best}}\} \leftarrow \{x_+, f_+, \Delta_+\}$, $\Delta_{\text{trial}} \leftarrow \Delta_{\text{best}}$.
1. Compute $x_{\text{trial}} \leftarrow x_{\text{best}} + \Delta_{\text{trial}}\, d$ and evaluate $f_{\text{trial}} = f(x_{\text{trial}})$ *(where d is "my" direction).*
2. Set $\{x_+, f_+, \Delta_+\} \leftarrow \{x_{\text{trial}}, f_{\text{trial}}, \Delta_{\text{trial}}\}$.
3. If $f_+ < f_{\text{best}}$, then $\{x_{\text{best}}, f_{\text{best}}, \Delta_{\text{best}}\} \leftarrow \{x_+, f_+, \Delta_+\}$, $\Delta_{\text{trial}} \leftarrow \Delta_{\text{best}}$, and *broadcast* the new minimum triplet $\{x_{\text{best}}, f_{\text{best}}, \Delta_{\text{best}}\}$ to all other processors. Else $\Delta_{\text{trial}} \leftarrow \frac{1}{2}\Delta_{\text{trial}}$.
4. If $\Delta_{\text{trial}} > tol$, goto Step 0. Else *broadcast* a local convergence message for the pair $\{x_{\text{best}}, f_{\text{best}}\}$.
5. Wait until either (a) enough of processes have converged for this point or (b) a better point is received. In case (a), exit. In case (b), goto Step 0.

Figure 4: Peer-to-peer version of APPS

incoming one if it is an improvement. If neither its own trial point nor any incoming messages are better, it performs a contraction and continues. Convergence is a trickier issue than in the synchronous version because the processors do not reach $\Delta_{\text{trial}} < tol$ at the same time. Instead, each processor converges in the direction that it owns, and then waits for the other processes to either converge to the same point or produce a better point. Since every good point is broadcast to all the other process, every process eventually agrees on the best point.

The final APPS algorithm is slightly different from the version in Fig. 4 because we spawn the objective function evaluation as a *separate* process. Our motivation is that we may sometimes want to stop an objective function evaluation before it completes in the event that a good point is received from another processor. We create a group of APPS *daemon* processes that follow the basic APPS procedure outlined in Fig. 4 except that each objective function evaluation will be executed as a separate process. The result is APPS daemons working in peer-to-peer mode, each owning a single *slave* objective function evaluation process.

The APPS daemon (see Fig. 5) works primarily as a message processing center. It receives three types of messages: a "return" from its spawned objective function evaluation and "new minimum" and "convergence" messages from APPS daemons.

When the daemon receives a "return" message, it determines if its current trial point is a new minimum and, if so, broadcasts the point to all other processors. The Δ_{trial} that is used to generate the new minimum is saved and can then be used to determine how far to step along the search direction. The alternative would be to reset $\Delta_{\text{trial}} = \Delta_0$ every time a switch is made to a new point, but then scaling information is lost which may lead to unnecessary additional function evaluations.

In the comparison of the trial and best f-values, we encounter an important caveat of heterogeneous computing [2]. The comparison of values (f's, Δ's, etc.) controls the flow of the APPS method, and we depend on these comparisons to give consistent results across processors. 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 $a = b$ if

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

where ϵ^*_{mach} is the maximum of all ϵ_{mach}'s.

A "new minimum" message means that another processor has found a point it thinks is best, and the receiving daemon must decide if it agrees. In this case, we must decide how to handle tie-breaking in a consistent manner. If $f_+ = f_{\text{best}}$, then we need to be able to say which point is "best" or if indeed the points we are comparing are equal (i.e., $x_{\text{best}} = x_+$). The tie breaking scheme is the following. If $f_+ = f_{\text{best}}$, then compare Δ_+ and Δ_{best} and select the larger value of Δ. If the Δ values are also equal, check next to see if indeed the two points are the same, but rather than comparing x_{best} and x_+ directly by measuring some norm of the difference, use a unique identifier included with each point. Thus, two points are equal if and only if

- **Return from Objective Function Evaluation.** Receive f_{trial}.

   1. Update x_{best} and/or Δ_{trial}.
      (a) If $f_{\text{trial}} < f_{\text{best}}$, then
         i. $\{x_{\text{best}}, f_{\text{best}}, \Delta_{\text{best}}\} \leftarrow \{x_{\text{trial}}, f_{\text{trial}}, \Delta_{\text{trial}}\}$.
         ii. *Broadcast* **new minimum** message with the triplet $\{x_{\text{best}}, f_{\text{best}}, \Delta_{\text{best}}\}$ to all other processors
      (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 $\Delta_{\text{trial}} > tol$, compute $x_{\text{trial}} \leftarrow x_{\text{best}} + \Delta_{\text{trial}} d$ and *spawn* a new objective function evaluation.
      (b) Else *broadcast* **convergence message** with $\{x_{\text{best}}, f_{\text{best}}, \Delta_{\text{best}}\}$ to all processors *including* myself.

- **New Minimum Message.** Receive the triplet $\{x_+, f_+, \Delta_+\}$.

   1. If $f_+ < f_{\text{best}}$, then
      (a) If $\Delta_+ > \Delta_{\text{best}}$ or I am *locally converged*, then flag $\leftarrow$ TRUE, else flag $\leftarrow$ FALSE.
      (b) Set $\{x_{\text{best}}, f_{\text{best}}, \Delta_{\text{best}}\} \leftarrow \{x_+, f_+, \Delta_+\}$.
      (c) If flag is TRUE, then *break* current objective function evaluation spawn, compute $x_{\text{trial}} \leftarrow x_{\text{best}} + \Delta_{\text{trial}}\, d$, and *spawn* a new objective function evaluation.

- **Convergence Message.** Receive the triplet triplet $\{x_+, f_+, \Delta_+\}$.

   1. Go though steps for **new minimum** to be sure that this point is x_{best}.
   2. Then, if *I* am the *temporary master* consider all the processes that have so far converged to x_{best}. If enough other processes have converged so that their associated directions form a positive spanning set, then output the solution, *shutdown* the remaining APPS daemon processes, and exit.

Figure 5: APPS Daemon Message Types and Actions

their f-values, Δ-values, and unique identifiers match.[5]

In certain cases, the current objective function evaluation is terminated in favor of starting one based on a new best point. Imagine the following scenario. Suppose three processes, A, B, and C start off with the same value for x_{best}, generate their own x_{trial}'s, and spawn their objective function evaluations. Each objective function evaluation takes several hours. Process A finishes its objective function evaluation before any other process and does not find improvement, so it contracts and spawns a new objective function evaluation. A few minutes later, Process B finishes its objective function evaluation and finds improvement. It broadcasts its new minimum to the other processes. Process A receives this message and terminates its current objective function evaluation process in order to move to the better point. This may save several hours of wasted computing time. However, Process C, which is still working on its first objective function evaluation, waits for that to complete before considering moving to the new x_{best}.

When the daemon receives a "convergence" message, it records the converged direction, and possibly checks for convergence. The design of the method requires that a daemon cannot locally converge to a point until it has evaluated at least one trial point generated from that best point along its search direction. Each point has an associated boolean *convergence table* which is sent in every message. When a process locally converges, it adds a TRUE entry to its spot in the convergence table before it sends a convergence message. In order to actually check for convergence of a sufficient number of processes, it is useful to have a *temporary master* to avoid redundant computation. We define the temporary master to be the process with the lowest process id. While this is usually process 0, it is not always the case if we consider faults, which are discussed in the next section. The temporary master checks to see if the converged directions form a positive spanning set, and if so outputs the result and terminate the entire computation.

Checking for a positive spanning set is done as follows. Let $\mathcal{V} \subset \mathcal{D}$ be the candidate for a positive basis. 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 [24]) *A spanning set $\mathcal{V} = \{v_1, v_2, \ldots, v_m\}$ is a positive spanning set if the maximum of the following LP is 1.*

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

[5] This system will miss two points that are equal but generated via different paths.

In the first case, we can use software for the nonnegative least squares problem from Netlib due to Lawson and Hanson [14]. In the second case, the software implementation is more complicated since we need both a QR factorization and a linear program solver, the latter of which is particularly hard to come by in both a freely available, portable, and easy-to-use format.

4 Fault Tolerance in APPS

The move toward a variety of computing environments, including heterogeneous distributed computing platforms, brings with it an increased concern for fault tolerance in parallel algorithms. The large size, diversity of components, and complex architecture of such systems create numerous opportunities for hardware failures. Our computational experience confirms that it is reasonable to expect frequent failures. In addition, the size and complexity of current simulation codes call into question the robustness of the function evaluations. In fact, application developers themselves will testify that it is possible to generate input parameters for which their simulation codes fail to complete 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 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 complete, 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 for more efficient and elegant fault tolerance. This algorithm-dependent variety of fault tolerance has already received a considerable amount of attention in the scientific computing community; see, e.g., [11, 12]. 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.

There are three scenarios that we consider when addressing fault tolerance in APPS: 1) the failure of a function evaluation, 2) the failure of an APPS daemon, and 3) the failure of a host. These scenarios are shown in Figure 6. The approaches for handling daemon and host failures are very similar to one another, but the function evaluation failure is treated in a somewhat different manner. When a function evaluation fails, it is respawned by its parent APPS daemon. If the failure occurs more than a specified number of times at the same trial point, then the daemon itself fails.[6] If an APPS daemon fails, the first thing the temporary master does is check for convergence since the now defunct daemon may have been in the process of that check when it died. Next it checks whether or not the directions owned by the remaining daemons form a positive basis. If so, convergence is still guaranteed, so nothing is done. Otherwise, all dead daemons are restarted. If a host fails, then the APPS daemons that were running on that host are restarted on a different host according to the rules stated for daemon failures. The faulty host is then removed from the list of viable hosts and is no longer used.

- **Exit from Function Evaluation.**
  1. If the number of tries at this point is less than the maximum allowed number, respawn the function evaluation.
  2. Else shutdown this daemon.
- **An APPS Daemon Failed.**
  1. Record failure.
  2. If I am the (temporary) master, then
     (a) Check for convergence and, if converged, output the result and terminate the computation.
     (b) If the directions corresponding to the remaining daemons do not form positive spanning set, respawn all failed daemons.
- **A Host Failed.**
  1. Remove host from list of available hosts.

Figure 6: Fault Tolerance Messages and Actions

Two important points should be made regarding fault tolerance in APPS. First, there are no single points of failure in the APPS algorithm itself. While there are scenarios requiring a master to coordinate efforts, this master is not fixed If it should fail while performing its tasks, another master 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, which has a single point of failure at the master PVM daemon [9]. We expect Harness [1], the successor to PVM, to eliminate this disadvantage. The second point of interest is that

[6]This situation can be handled in different ways for different applications; attempts to evaluate a certain point could be abandoned without terminating the daemon.

no checkpointing or replication of processes is necessary. The algorithm reconfigures on the fly, and new APPS daemons require only a small packet of information from an existing process in order to take over where a failed daemon left off. Therefore, we have been able take advantage of characteristics of APPS in order to elegantly incorporate a high degree of fault tolerance with very little overhead.

Despite the growing concern for 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, APPS can recover from the failure of any type of process, including the failure of a temporary master, on its own and has no checkpointing whatsoever.

5 Numerical Results

We compare PPS[7] and APPS 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 nodes 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: broyden2a, broyden2b, chebyquad, epowell, toint_trig, and vardim [18, 5]. Since the function evaluations are extremely fast, we added extra "busy work" in order to slow them down to better simulate the types of objective functions we are interested in.[8]

The parameters for APPS and PPS were set as follows. Let $n = 4$ be the problem dimension, and let p be the number of processors. The first $2n$ search directions are $\{e_1, e_2, \ldots, e_n, -e_1, -e_2, \ldots, -e_n\}$. The remaining $p - 2n$ directions are vectors that are randomly generated (with a different seed for every run) and normalized to unit length. This set of search directions is a positive spanning set. We initialize $\Delta = 1.0$ and $tol = 0.001$.

[7] We are using our own implementation of a positive basis PPS, as outlined in Fig. 3, rather than the well-known parallel direct search (PDS) [22]. PDS is not based on the positive basis framework and is quite different from the method described in Fig. 3, making comparisons difficult.

[8] More precisely, the "busy work" was the solution of a 100×101 nonnegative least squares problem.

We added two additional twists to the way Δ is updated for all tests. First, if the same search direction yields the best point two times in a row, Δ is doubled before the broadcast. Second, the smallest allowable Δ for a "new minimum" is such that at least three contractions will be required before local convergence. That way, we are guaranteed to have several evaluations along each search direction for each point.

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	*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	*235*	*N/A*	*0.22*	*6.10*	*30.63*

Table 1: Detailed results for epowell on eight processors.

Before considering the summary results, we examine detailed results from two sample runs given in Table 1. 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 varies for each process, in this case by as much as 25%. Furthermore, APPS sometimes "breaks" functions midway through execution. On the other hand, every process in PPS executes the same number of function evaluations, and there are no breaks. For both APPS and PPS, the initialization time is longer for the first process since it is in charge of spawning all the remaining tasks. The idle time varies from task to task but is overall much lower for APPS than PPS. An APPS process is only idle when it is locally converged, 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 is the average over all processes except in the case of total time, in which case the maximum over all times is reported.

Because some of the search directions are generated randomly, every run of PPS and APPS generates a different path to the solution and possibly different solutions in the case of multiple minima.[9] Because of the nondeterministic nature of APPS, it gets different results every run even when the search directions are identical. Therefore, for each problem we report average summary results from 25 runs.

Problem Name	Procs	Function Evals APPS	Function Evals PPS	APPS Breaks	Idle Time APPS	Idle Time PPS	Total Time APPS	Total Time PPS
broyden2a	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
broyden2b	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
chebyquad	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
epowell	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
toint_trig	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
vardim	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 2: Results on a collection of four dimensional test problems.

The test results are summarized in Table 2. These tests were run in a fairly favorable environment for PPS—a cluster of homogeneous, dedicated processors. The

[9] The exception is PPS with $n = 8$. Because there are no "extra" search directions, the solution to the path is the same for every run—only the timings differ.

primary difficulty for PPS is the cost of synchronization in the global reduction. In terms of average function evaluations per processor, both APPS and PPS 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 expect 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 22 of 24 cases. The total time for APPS either stayed steady or reduced as the number of processors increased. In contrast, the total PPS time increased as the number of processors increased due to the synchronization penalty.

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

5.2 TWAFER: A Thermal Design Problem

This engineering application concerns the simulation of a thermal deposition furnace for silicon wafers. The furnace contains a vertical stack of 50 wafers and several heater zones. The goal is to achieve a specified constant temperature across each wafer and throughout the stack. The simulation code, TWAFER [10], yields measurements at a discrete collection of points on the wafers. The objective function f is defined by a least squares fit of the N discrete wafer temperatures T_j to a prescribed ideal T_* as follows

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

where x_i is the unknown power parameters for the heater in zone i. We consider the four and seven zone problems.

For this problem, 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. We set $\Delta = 10.0$ and $tol = 0.1$.

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. The input file for TWAFER must contain an entire description of the furnace and the wafers. We are only changing a few values within that file, so our input filter generates the input file for TWAFER by using a "template" input file. This template file contains tokens that are replaced by our optimization variables. The output file from TWAFER contains the heat measurements at discrete points. Our output filter reads in these values and computes the least squares difference between these and the ideal temperature in order to determine the value of the objective function.

An additional caveat is that TWAFER must be executed in a uniquely named subdirectory so that its input and output files are not confused with those of any other TWAFER process that may be accessing the same disk.

Lastly, 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 it has nonnegativity constraints on the power settings. We use a simple barrier function that returns a large value (e.g., 10^{50}) if $x_i < 0$ for any i.

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 3: Results on the four and seven zone TWAFER problems.

Results for the TWAFER problem are given in Table 3. The four zone results are the averages over ten runs, and the seven zones results are averages over nine runs. (The tenth PPS run failed due to a node fault. The tenth APPS run had several faults, and although it did get the final solution, the summary data was incomplete.) Here we also list the value of the objective function at the solution. Observe that PPS yields slightly better function values (compared to the original value of more than 1000) on average but at a cost of more function evaluations and more time. The average function evaluation execution time for the four zone problem is 1.3 seconds and for the seven zone problem is 10.4 seconds; however, the number of function evaluations includes instances where the bounds were violated in which case the TWAFER code was not executed and the execution time is essentially zero since we simply return 10^{50}. Once again PPS has a substantial amount of idle time. The relatively high APPS idle time in the seven zone problem was due to a single run in which the idle time was particularly high for some nodes (634 seconds on average).

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 parameters. The objective function is defined as

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

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 [19] 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 APPS variables. 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 consists 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. 7). 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.

The APPS parameters were set as follows. The search directions were generated in the same way as those for the test problems. We set $\Delta = 1.0$ and $tolis 0.1$ (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.

The results from APPS and PPS on the SPICE problem are reported in Table 4. In this case, we are reporting the results of single runs, and we give results for 34 and 50 processors. The average SPICE run time is approximately 20 seconds; however, we once again 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.

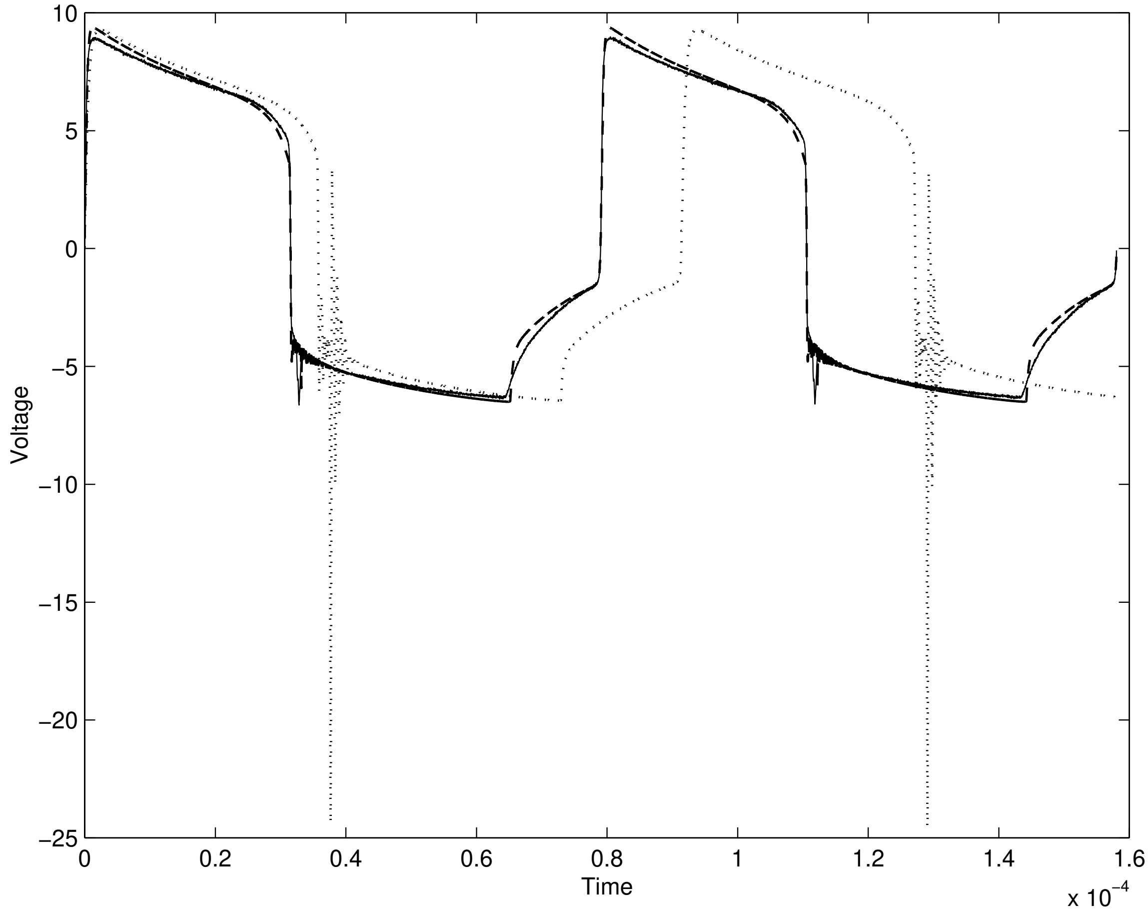

Figure 7: Spice results. The solid line represents the experimental output. The dashed line represents the simulation output after optimization. The dotted line represents the starting point for the optimization.

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 4: Results for the 17 variable SPICE problem.

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

Table 5: APPS results for the 17 variable SPICE with a failure approximately every 30 seconds.

Table 5 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 daemons 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%, 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 for convergence (see Lewis and Torczon [17]).

6 Conclusions

The newly-introduced APPS method is superior to PPS in terms of overall computing time on a homogeneous cluster environment for both generic test problems and engineering applications. We expect the difference to be even more pronounced for larger problems (both in terms of execution time and number of variables) and for heterogenous cluster environments. Unlike PPS, APPS does not have any required synchronizations and, thus, gains most of its advantage by reducing idle time.

APPS is fault tolerant and, as we see in the results on the SPICE problem for 34 processors, does not suffer much slow-down in the case of faults.

In forthcoming work, Kolda and Torczon [13] will show that in the unconstrained case the APPS method converges (even in the case of faults) under the same assumptions as pattern search [23].

Although the engineering examples used in this work have bound constraints, the APPS method was not fully designed for this purpose, as evidenced in the poor results on the SPICE problem with faults on 50 processors. Future work will explore the algorithm, implementation, and theory in the constrained cases.

In the implementation described here, the daemons and function evaluations are in pairs; however, for multi-processor (MPP) compute nodes, this means there will be several daemon/function evaluation pairs per node. An alternative implementation of APPS is being developed in which there is exactly one daemon per node regardless of how many function evaluations are assigned to it. As part of this alternative implementation, the ability to dynamically add new hosts as they become available (or to re-add previously failed hosts) will be incorporated.

Another improvement to the implementation will be the addition of a function value cache in order to avoid reevaluating the same point more than once. The challenge is deciding when two points are actually equal; this is especially difficult without knowing the sensitivity of the function to changes in each variable.

The importance of positive bases in the pattern raises several interesting research questions. First, we might consider the best way to generate the starting basis. We desire a pattern that maximizes the probability of maintaining a positive basis in the event of failures. Another research area is the affect that "conditioning" of the positive basis has on convergence. Our numerical studies have indicated that the quality of the positive basis may be an issue. Last, supposing that enough failures have occurred so that there is no longer a positive basis, we may ask if we can easily determine the fewest number of vectors to add to once again have a positive basis. Our current implementation simply restarts all failed processes.

A Acknowledgments

Thanks to Jim Kohl, Ken Marx, Juan Meza for helpful comments and advice in the implemenation of APPS and the test problems.

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*. Submitted to International Journal on Future Generation Computer Systems, 1999.

[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*, Tech. Rep. Mathematical Programming 99–05, Computer Sciences Department, University of Wisconsin, Madison, Wisconsin, 1999.

[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*, Research Report CS–86–45, Faculty of Mathematics, University of Waterloo, Waterloo, Canada, 1986.

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

[9] 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.

[10] 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.

[11] Y. Kim, J. S. Plank, and J. Dongarra, *lgorithm-based diskless checkpointing for fault tolerant matrix operations*, in 25th International Symposium on Fault-Tolerant Computing, Pasadena, CA, June, 1995.

[12] ——, *Fault tolerant matrix operations for networks of workstations using multiple checkpointing*, in HPC–Asia 97, April 1997.

[13] T. G. Kolda and V. Torczon, *On the convergence of asynchronous parallel direct search.* In preparation, 1999.

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

[15] R. M. Lewis, V. Torczon, and M. W. Trosset, *Why pattern search works*, Optima, (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, Mail Stop 132C, NASA Langley Research Center, Hampton, Virginia 23681–2199, 1996.

[17] ——, *Pattern search methods for linearly constrained minimization*, Tech. Rep. 98–3, Institute for Computer Applications in Science and Engineering, Mail Stop 132C, NASA Langley Research Center, Hampton, Virginia 23681–2199, January 1998. To appear in SIAM Journal on Optimization.

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

[19] 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, California 94720, May 1993.

[20] 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.

[21] 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.

[22] 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, Texas 77005–1892, 1992.

[23] ——, *On the convergence of pattern search algorithms*, SIAM Journal on Optimization, 7 (1997), pp. 1–25.

[24] S. E. Wright, *A note on positively spanning sets.* To appear in *American Mathematical Monthly*, 1999.