Parallel Parameter Tuning for Applications with Performance Variability

Vahid Tabatabaee
Dept. of Computer Science,
University of Maryland,
College Park, MD 20742
vahid@cs.umd.edu

Ananta Tiwari
Dept. of Computer Science,
University of Maryland,
College Park, MD 20742
tiwari@cs.umd.edu

Jeffrey K. Hollingsworth
Dept. of Computer Science,
University of Maryland,
College Park, MD 20742
hollings@cs.umd.edu

ABSTRACT

In this paper, we present parallel on-line optimization algorithms for parameter tuning of parallel programs. We employ direct search algorithms that update parameters based on real-time performance measurements. We discuss the impact of performance variability on the accuracy and efficiency of the optimization algorithms and propose modified versions of the direct search algorithms to cope with it. The modified version uses multiple samples instead of single sample to estimate the performance more accurately.

We present preliminary results that the performance variability of applications on clusters is heavy tailed. Finally, we study and demonstrate the performance of the proposed algorithms for a real scientific application.

1. INTRODUCTION

Software today makes extensive use of libraries that are hard to tune to specific application requirements. The problem becomes even more complex when we realize that the application requirements and characteristics are functions of the application input, computing system architecture, and other applications running simultaneously on the same system. Under these conditions, it is practically impossible to manually tune the parameters and optimize the application performance; therefore, on-line automated parameter tuning is the only plausible solution. In addition, on most cluster based parallel computing platforms, program execution time is somewhat variable, due to the activities such as house keeping processes running, and other transient disruptions.

Active Harmony [18] is an infrastructure that provides a mechanism for applications to become tunable with minimal changes to the application structure. The user provides Active Harmony with a list of the tunable parameters, and their type and range. Active Harmony iteratively runs the program, monitors its performance (running time) and tunes the parameters to optimize the performance.

SC|05 November 12-18, 2005, Seattle, Washington, USA

In this paper, we focus on the design and characteristics of suitable optimization algorithms for on-line tuning systems such as Active Harmony. On-line tuning problems have some distinct characteristics and requirements that must be considered when designing the optimization algorithm.

The characteristics that we consider and discuss in this paper are:

1-Performance Metrics: The final result and the convergence time (in number of evaluation of the objective function) are two common performance metrics for optimization algorithms. However, for the on-line tuning problem, these are not the most appropriate ones. In the on-line tuning, the overall performance of the system from the start to the end are equally important for us. Therefore, the appropriate performance metric should consider and capture the transient behavior and performance of the intermediate visited points in the path to the final solution as well.

2-Unstructured Optimization Space: Many of the optimization algorithms work appropriately on well structured problems. For instance, gradient based algorithms are appropriate for continuous convex optimizations. In Harmony, we are often dealing with integer restricted parameters and an unknown non-convex function with multiple local minimums. Therefore, we have to resort to a class of optimization algorithms that work under these conditions.

3-Parallel Processing: We are primarily interested in high performance computing applications, where multiple instances of the same code are simultaneously running on multiple processors (i.e. SPMD problems). In this case, it is desirable to employ parallel optimization algorithms that can take advantage of the underlining parallel structure to converge faster.

4-Performance Variability: The on-line tuning process monitors performance of the application for different parameter values and based on the observed performances modifies parameters. The goal is to ultimately find the parameter values that has optimal, or at least, near optimal performance. Most algorithms are designed and studied assuming perfect and accurate monitoring, which is not the case in real systems. In other words, the observed performance for

a fixed set of parameters is not always the same. Therefore, it is desirable to develop algorithms that are resilient and able to converge to good solutions, even in the presence of performance variability.

The contribution of this paper are: (1) *Parallel* optimization algorithms that are *resilient* to performance variability, (2) a simple *stochastic* model and monitoring mechanism for performance variability in HPC systems. We present a simple two priority queue system to model the performance variability. Based on this model, we propose to use multiple observations with a simple minimization operator to estimate the performance more accurately. We show that the minimization operator is very effective, even when the performance variability is a heavy tail stochastic process with infinite variance.

The rest of the paper is organized as follows: Section 2 defines the problem and introduces the performance metrics for the optimization algorithms. Section 3 describes the direct search algorithms for optimization and introduces the parallel algorithms that we developed for on-line tuning. Section 4 elaborates on stochastic modeling of the performance variability. Section 5 describes how to use multiple sampling to cope with performance variability, and properties of the minimization operator. We present the simulation results and discuss them in section 6. Section 7 reviews related works, and section 8 concludes the paper.

2. PROBLEM DEFINITION AND PERFORMANCE METRIC

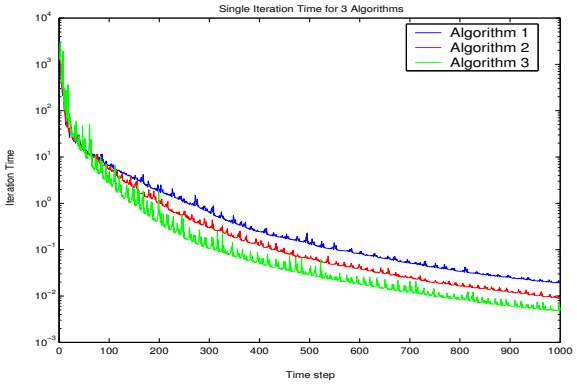
We consider software applications with an iterative structure using SPMD style computation. After finishing each iteration on all processors, information is exchanged between nodes and the next iteration starts. We consider that the application should run for a fixed number of iterations or time steps, say K, to get the ultimate result. Our objective is to minimize $Total\_Time(K)$, which is the time that it takes to run the program for the desired number of time steps, K. Suppose that we want to run the application for K time steps on P parallel processors. Let T_k be the time that it takes to run the kth iteration, and $t_{p,k}$ be the time that it takes to run kth iteration on the pth processor. We have,

$$T_k = \max_{p=1,\cdots,P}(t_{p,k}), \tag{1}$$

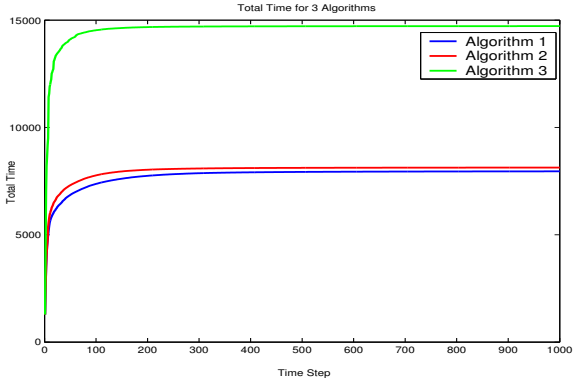
and

$$Total\_Time(K) = \sum_{k=1}^{K} T_k. \tag{2}$$

Our main performance metric is $Total\_Time(K)$. Two typical performance metrics for optimization algorithms are the final solution after convergence and the convergence speed. These asymptotic metrics could be misleading for on-line tuning, since the number of iterations is fixed to K and the algorithm may not have enough time to converge. Even if K is large enough that the algorithm converges, the overall performance could be dominated by the performance of the algorithm at the initial time steps before convergence.



(a) Single Iteration Time plot for 3 algorithms



(b) Total Time plot for 3 algorithms

Figure 1: Single Iteration time and Total Iteration Time for 3 different algorithms. Fig.1-(a) shows the Iteration Time, where it is clear that Algorithm 3 converges to a better solution ultimately. Fig.1-b shows the Total Time from which it is clear that Algorithm 1 performs better.

Furthermore, for every iteration k, the worst case performance (maximum value) is used for T_k. This is again different from common practice, where the best performance is considered in every iteration. Note that in our on-line tuning model all processors should wait for the last processor before starting the next iteration; hence, the worst case performance is the bottle neck. [1]

To clarify the performance metric issues, consider three optimization algorithms whose performance are plotted in Fig.1. These algorithms are different variants of direct search optimization methods that we discuss later. Fig.1-a shows each iteration's worst case performance, T_k, versus iteration. This plot is closer to the plots that are typically used for comparison of the optimization algorithms, where the best case performance is usually used. However, for the on-line tuning the Total Time plot of Fig.1-b is more appropriate. Note that the Total Time curve of Fig.1-b is the integral of the worst case performance given in Fig.1-a. By looking at Fig.1-a, one may conclude that Algorithm 3 is the best, where as from Fig.1-b, we can conclude that Algorithm 1 is more appropriate for on-line tuning. The main reason

[1]Our actual tuning system works for applications that do not have this synchronization requirement, but for the purpose of algorithm analysis it is useful to include.

behind the discrepancy is the transient behavior of Algorithm 1 in the first 100 time steps, where it has significant fluctuations.

In summary, for on-line tuning, initial transient behavior of the algorithm can be more significant than the final value at the convergence point of the algorithm. This fact should be taken into account in selecting the appropriate optimization algorithm. For instance, randomized scheduling algorithms such as Simulated Annealing and Genetic Algorithms that are often considered suitable for unstructured optimization problems are not appropriate for on-line tuning, since even though they can ultimately converge to the optimal solution, they have very poor initial performance.

3. DIRECT SEARCH ALGORITHMS

Direct search methods are a class of optimization algorithms that do not explicitly use function derivatives. Consider the problem of finding a local minimum of a real-valued function $f(X)$. If f is differentiable and $\nabla f(X)$ can be computed or estimated, there is a plethora of gradient-based algorithms to solve this problem. However, if $\nabla f(X)$ is not available or if f is not differentiable, we have to rely on alternative algorithms such as direct search methods. The on-line tuning problem is a very good example for the latter group, since in most cases the performance function is not differentiable, and if it is differentiable, its gradient is not explicitly computable.

The Nelder and Mead Simplex algorithm [13] is one of the most commonly used direct search methods. In fact, the Simplex algorithm has been used in the Active Harmony system [18]. Despite its popularity, the Simplex algorithm has several shortcomings for on-line tuning applications, which motivated us to consider Rank Ordering algorithms. The Rank Ordering algorithms are an alternative group of direct search algorithms. In the following we briefly review the Simplex and Rank Ordering algorithms, and then describe our Parallel Rank Ordering implementation for on-line tuning.

3.1 Simplex Algorithm

The Nelder-Mead Simplex method is a direct search algorithm for minimizing a function of multiple variables. For a function of N variables, the algorithm maintains a set of $N+1$ points forming the vertices of a simplex or polytope in N-dimensional space. This simplex is successively updated at each iteration by discarding the vertex having the highest function value and replacing it with a new vertex having a lower function value.

Let $v_0, \cdots, v_N$ be the vertices of the corresponding simplex and v_N be the point with the highest function value among them. We first compute c, the centroid of the other simplex vertices:

$$c = \left(\sum_{i=0}^{N-1} v_i \right) / N \tag{3}$$

The point v_N will be replaced by a point on the line $v_N + \alpha(c - v_N)$. Typically, $\alpha \in \{0.5, 2, 3\}$, but the α selection process depends on implementation. Usually, the first step is to compute the function value at the reflection point ($\alpha = 2$). Depending on the result we either check for expansion ($\alpha = 3$) or contraction ($\alpha = 0.5$). If none of the computed values is less than the function value at v_N, we contract the whole simplex around the best point.

The simplex algorithm works well, when there is only 1 tunable parameter [12]. However, its performance becomes unpredictable when the number of variables increases. One of the basic problems with the Simplex algorithm is the ability to deform the simplex, which may eventually lead to a degenerate simplex that does not span the N-dimensional space. Furthermore, the Simplex algorithm is inherently a sequential algorithm and is not able to fully take advantage of parallel infrastructures. In the on-line tuning applications that we consider, often there are multiple processors that are executing the same or very similar code and after each iteration they exchange information. Therefore, it is desirable to use different parameter values on different processors and evaluate their performance concurrently. Rank Ordering algorithms can take advantage of the concurrent performance evaluation to speedup convergence.

In summary the major problems with the Simplex algorithm for on-line tuning are:

1. It may converge to a degenerate simplex; therefore, it would not be able to search the parameter space effectively.
2. Its performance is very sensitive to the initialization and implementation details.
3. It can not fully take advantage of parallel infrastructure to speedup the process.
4. It is not stable in the presence of performance variability and measurement errors.

3.2 Rank Ordering Algorithm for Parameter Tuning

Our discussion on the Simplex algorithm clarifies that not all the direct search algorithms reliably converge to a local minimum. The Simplex algorithm can sometimes work efficiently, but it can also fail unpredictably. Kolda, et al. [7] introduces a class of reliable direct search algorithms, Generating Set Search (GSS). they also prove that if the objective function f is continuously differentiable, then GSS produces a sequence of points X_k such that,

$$\lim_{k \to \infty} \inf \|\nabla f(X_k)\| = 0. \tag{4}$$

This result is similar to the convergence results for gradient based algorithms, since it guarantees that the GSS algorithms converge to a stable point of the objective function f. Therefore, compared to non-GSS direct search methods, the GSS algorithms have predictable and more reliable performance.

In this paper, we use Rank Ordering direct search algorithms [11], which are in the GSS class and constitute all necessary conditions for convergence. The Rank ordering algorithms

Algorithm 1 : Sequential Rank Ordering

1: Start with initial simplex with vertices $\{v_0^0, \cdots, v_0^n\}$ and evaluate $f(v_0^j), \quad j = 0, \cdots, n$
2: $k = 0$
3: **while** Stopping criteria not valid **do**
4: Reorder simplex vertices so that $f(v_k^0) \leq \cdots \leq f(v_k^n)$
5: $r_k = 2v_k^0 - v_k^n$, evaluate $f(r_k)$ {*Reflection checking step*}
6: **if** $f(r_k) < f(v_k^0)$ **then**
7: $e_k = 3v_k^0 - 2v_k^n$, evaluate $f(e_k)$ {*Expansion checking step*}
8: **if** $f(e_k) < f(r_k)$ **then** {*Accept expansion*}
9: $v_{k+1}^j = 3v_k^0 - 2v_k^j$, evaluate $f(v_{k+1}^j) \quad j = 1, \cdots, n$ {*Expansion steps*}
10: **else** {*Accept reflection*}
11: $v_{k+1}^j = 2v_k^0 - v_k^j$, evaluate $f(v_{k+1}^j) \quad j = 1, \cdots, n$ {*Reflection steps*}
12: **end if**
13: **else** {*Accept shrink*}
14: $v_{k+1}^j = 0.5(v_k^0 + v_k^j)$, evaluate $f(v_{k+1}^j) \quad j = 1, \cdots, n$ {*Shrink steps*}
15: **end if**
16: k = k+1
17: **end while**

can leverage parallelism [5] to speedup convergence. For clarity and brevity, we explain Rank Ordering algorithms in the basic form appropriate for our application and not in the most general form.

Rank Ordering algorithms start with an initial simplex with vertices $\{v_0^0, \cdots, v_0^n\}$ and evaluate f at all vertices. The initial simplex should span the optimization space, hence $n \geq N$. At every iteration, the simplex is either reflected, expanded or shrunk around its best vertex (the vertex with the least function value).

Algorithm 1 is the basic Sequential Rank Ordering (SRO) algorithm given in [11]. At each iteration, one of the three possible steps of reflection, expansion, or shrink will be accepted. Examples for each one of these steps are shown in Fig. 2 for a 2-dimensional space and a 3 point simplex.

At each iteration r_k, the reflection point of the worst vertex, v_k^n around the best vertex, v_k^0 is computed. Intuitively, the direction from the point with lowest function value to the point with highest function value approximates the gradient direction. If $f(r_k)$ is less than $f(v_k^0)$, i.e., the *best* performance point on the simplex, we compute e_k, the expansion of the worst point. If the expansion point performance is better than the reflection point, we accept the expansion, otherwise we accept the reflection. Note that reflection and expansion are only accepted if their performance is better than the *best* point discovered so far (otherwise we shrink the simplex). This is different from the Simplex algorithm approach, where the reflection is accepted if its performance is better than the *worst* vertex of the simplex.

We considered and tested alternative parallel variants of the SRO algorithm. However, for brevity, we only explain the version selected, which had the best overall performance.

The proposed Parallel Rank Ordering (PRO) for on-line tuning is given in Algorithm 2. Function $\Pi(.)$ that is used in

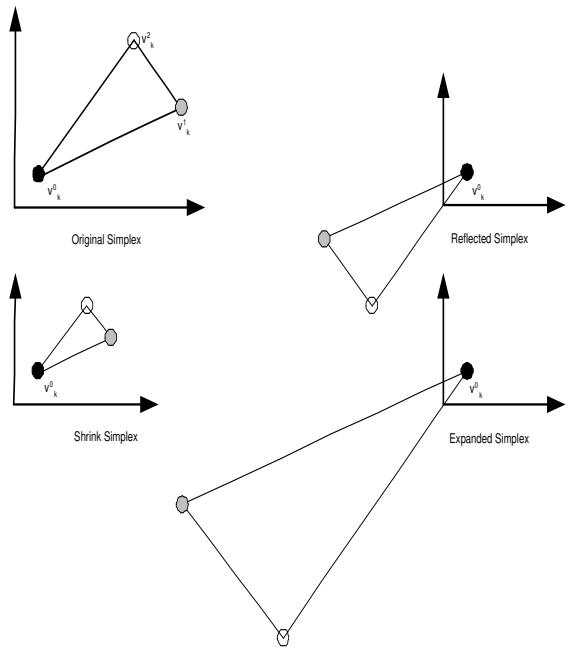


Figure 2: Original 3 point simplex in 2-dimensional space and the transformed simplexes after reflection, shrink and expansion around the point v_k^0

PRO description is projection mapping. We will give a precise definition for $\Pi(.)$ later, but its purpose is to make sure that the computed points always belong to the admissible region. After initialization steps the main loop starts in line 3. In the main loop, performance (function value) at all the reflection points are found in parallel on n processors (line 5). If reflection is successful, which means that there is at least one reflected point with better performance than the best point of the simplex, we check for expansion (line 8). Recall that in the SRO, we only compute one point in the reflection checking step (line 5 of Algorithm 1). Therefore, the PRO criteria for reflection and expansion, since it relies on performance of n points, is more reliable and improves the performance.

Before, computing all n expansion points in line 10, we check the outcome of expansion in line 8 for one point that has the highest chance. This seems to be counter-intuitive at first glance, since we are not taking full advantage of parallelism. However, in our simulations, we realized there are some expansion points with very poor performance that can slow down the algorithm. Therefore, to avoid these time consuming instances, we are better off to calculate the expansion point performance for the most promising case first, and if it is successful then perform the expansion for other points.

If reflection is not successful and there is no reflected point performing better than the best point of the simplex, shrinking of the simplex is accepted. All the shrinking points and their performance are computed in parallel.

If we have at least n parallel processors (we will talk later about the value of n), each iteration of the PRO algorithm

Algorithm 2 : Parallel Rank Ordering

1: Start with initial simplex with vertices $\{v_0^0, \cdots, v_0^n\}$ and evaluate $f(v_0^j), \quad j = 0, \cdots, n$ in parallel on n processor.
2: $k = 0$
3: **while** Stopping Criteria Not Valid **do**
4: Reorder simplex vertices, so that $f(v_k^0) \leq \cdots \leq f(v_k^n)$
5: Compute n reflection points $r_k^j = \Pi\left(2v_k^0 - v_k^n\right)$, and function values $f(r_k^j), \quad j = 1, \cdots, n$ in parallel on n processors. {*Reflection step*}
6: $l = \arg\min_j f(r_k^j)$
7: **if** $f(r_k^l) < f(v_k^0)$ **then**
8: $e_k = \Pi\left(3v_k^0 - 2v_k^l\right)$, evaluate $f(e_k)$ {*Expansion checking step*}
9: **if** $f(e_k) < f(r_k^l)$ **then** {*Accept expansion*}
10: Compute n expansion points $e_k^j = \Pi\left(3v_k^0 - v_k^n\right)$, and function values $f(e_k^j), \quad j = 1, \cdots, n$ in parallel on n processors. {*Expansion step*}
11: $v_{k+1}^j = e_k^j \quad j = 1, \cdots, n$
12: **else** {*Accept reflection*}
13: $v_{k+1}^j = r_k^j \quad j = 1, \cdots, n$
14: **end if**
15: **else** {*Accept shrink*}
16: Compute $\Pi\left(v_{k+1}^j = 0.5v_k^0 + 0.5v_k^j\right)$, and $f(v_{k+1}^j) \quad j = 1, \cdots, n$ in parallel on n processor. {*Shrink step*}
17: **end if**
18: k = k+1
19: **end while**

takes at most 3 time steps (reflection, expansion checking, and expansion steps). In the following, we will go over some PRO implementation issues.

3.2.1 Projection Operator

On-line tuning is a *constrained* optimization problem. Therefore, in each step we have to make sure that the computed points are admissible, i.e. they satisfy the constraints. The projection operator $\Pi(.)$ takes care of this problem by mapping points that are not admissible to admissible points. We consider two types of parameter constraints: boundary constraints and internal discontinuity constraints.

Boundary constraints are upper and/or lower limits for the parameters. If the computed value for a parameter is less (greater) than the lower (upper) limit, the projected value for that parameter would be equal to the lower (upper) limit.

Some tuning parameters can only have admissible discrete values. For instance, many of the variables are finite integer numbers. The projection operator makes sure that the computed parameters are rounded to an admissible discrete value. Consider the point $x = (x(1), \cdots, x(N)) \in R^N$ that is computed after a transformation (reflection, extraction, or shrink) around the point v_k^0 in PRO. For every parameter i, if $x(i)$ is admissible it will remain the same. Otherwise, if $l(i) < x(i) < u(i)$, for two consecutive admissible values $l(i)$ and $u(i)$, then projection of $x(i)$ is $l(i)$ if $v_k^0(i) < x(i)$, and is $u(i)$ if $v_k^0(i) > x(i)$. In other words, every parameter is rounded to its lower or higher discrete value, whichever is closer to the transformation center $v_k^0(i)$. In this way, after a finite number of consecutive shrinking transformations, all discrete parameters $x(k)$ become equal to $v_k^0(i)$. This property will be used to check convergence in the stopping criteria.

3.2.2 Stopping Criteria

After every iteration, the algorithm checks to see if all simplex vertices are the same (for discrete parameters) or very close (for continuous parameters). If that is the case, the algorithm checks for convergence. To that end, a 2N-dimensional simplex around the v_k^0 with vertices $\{v_k^0 + u_i e_i,\ v^0 - l_i e_i : i = 1, \cdots, N\}$ is generated, where e_i is the unit vector with 1 in its ith index and zero for other indexes. If ith parameter is continuous, u_i and l_i are sufficiently small numbers, and if it is discrete, u_i and l_i are selected such that $v_k^0(i) + u_i$ and $v_k^0(i) - l_i$ are discrete neighbors of $v_k^0(i)$. If $v_k^0(i)$ is a lower (upper) boundary value, then l_i (u_i) is zero.

Performance at all $2N$ generated points are computed; if none of them outperforms v_k^0, then v_k^0 is a local minimum and we can stop, otherwise we can continue PRO with the generated simplex.

3.2.3 Initial Simplex

The initial simplex needs to be non-degenerate so that it can span the whole parameter space; therefore it should have at least $N + 1$ vertices. However, for dealing with discrete parameters this may not be sufficient. In fact, this was confirmed in our simulations, where we observed a simplex with $2N$ vertices performs much better than a minimal simplex with $N + 1$ vertices. Vertices of the initial simplex in PRO are: $\{\Pi\left(c \pm b_i.e_i\right), \quad i = 1, \cdots, N\}$, where c is the center point of the admissible region.

Optimal value for vector $b = (b_1, \cdots, b_N)$ elements, which controls size of the initial simplex, is problem dependent. In general, if b is too small, the algorithm may trap in a low performance local minimum close to the center. On the other hand, if it is too large, performance may suffer from poor performance of marginal parameter values [3]. As a compromise, we set b_i to,

$$b_i = 0.1(u(i) - l(i)),$$

where $l(i)$ and $u(i)$ are the lower and upper limits for parameter i respectively. We plan in the future to investigate this and develop adaptive methods for computing b.

4. PERFORMANCE VARIABILITY AND ITS IMPACT ON PARAMETER TUNING

Besides the tunable parameters, there are many other factors affecting a program's performance. Therefore, even for a fixed set of tunable parameters, the application performance varies in time. Other applications running on the same processor, network performance, operating system, and memory architecture are common sources of performance variability. We are not attempting to minimize or control the variability here, but our goal is to design optimization algorithms that are resilient to performance variability. In this section, we first provide a simple stochastic model for performance variability and present some primary data, based on measurements from a real cluster, indicating the presence of a *heavy tail* component in the probability distribution function (pdf).

4.1 Two job Model

We model the computing system as a single machine with a strict priority scheduler serving two set of jobs. The tunable application is the second priority job and all sources of performance variability are modelled as the first priority job. The computing system processes (serves) the application, whenever there is no first priority request. First priority job arrival is a random process; therefore, the application performance (finishing time of the second priority job) is a random variable (r.v.).

Let $f(v)$ be the application performance for parameter v, when there is no first priority job in the system. The application performance, y is:

$$y = f(v) + n(v). \tag{5}$$

The random variable $n(v)$ is the time the system spends processing first priority jobs, while the application is in the system. It will shortly become clear why $n(\cdot)$ is a function of v in our model. Let ρ, the idle system throughput, be the system throughput when there is no second priority job in system. In other words, ρ is the average processing time of the first priority jobs. From the application perspective, the effective system throughput is $1-\rho$. The average (expected) system performance is:

$$E(y) = \frac{f(v)}{1-\rho}. \tag{6}$$

From equations (5)-(6) we have:

$$E(n(v)) = \frac{\rho}{1-\rho} f(v). \tag{7}$$

Now, it should be clear that the expected variability is a linear function of $f(v)$; hence, the r.v. $n(\cdot)$ is a function of the application parameters v.

4.2 Heavy Tail Model

In the previous section, using the two job model, we showed that the expected performance is a linear function of the idle system throughput. In this section, we go one step further and introduce an appropriate distribution for the performance. Derivation of an accurate model requires further research and study; however, at this stage, our goal is to capture those characteristics that are critical for the optimization process.

Previous studies of the performance variability indicate that there is a non-negligible probability of observing large variations in the finishing time of an application [9, 15]. This feature can be captured through the use of so-called *heavy tail* models. Heavy tail distributions exhibit tails that decay as a hyperbolic function, which is in contrast to the typical exponential decay in other models such as a Gaussian distribution.

A distribution is said to have a heavy-tail if:

$$P[X > x] \sim x^{-\alpha}, \text{as } x \to \infty, \quad 0 < \alpha < 2 \tag{8}$$

This means that regardless of the distribution for small values of the random variable, if the asymptotic shape of the distribution is hyperbolic, it is heavy-tailed [4]. The simplest heavy-tailed distribution is the Pareto distribution which is hyperbolic over its entire range and its cumulative distribution function (cdf) is given by:

$$F_X(x) = P[X \leq x] = 1 - (\beta/x)^{\alpha}, \tag{9}$$

where β is the smallest value the r.v. can take. For $1 < \alpha < 2$, Pareto distribution has finite mean and infinite variance, and for $0 < \alpha \leq 1$, both mean and variance are infinite.

Heavy-tailed distributions have properties that are qualitatively different from commonly used distributions such as Exponential, Normal or Poisson distributions. Therefore, it is important to figure out if the performance variability distribution is heavy tail. In the next section, we will try to answer this question.

4.3 Performance Variability for GS2 Application:

GS2 [6, 8] is a physics application program, developed to study low-frequency turbulence in magnetized plasma. In this section, we use GS2 to study the performance variability, when the application parameters are fixed. In section 6, we use GS2 to study performance of the PRO algorithm in tuning application parameters. GS2 has several tunable parameters, which can be set to represent the appropriate conditions for different modes. We considered three major parameters in our study: ntheta (number of grid points per 2 pi segment of field line), negrid (energy grid), and nodes (number of nodes).

Fig. 3 shows the running time of the GS2 with *fixed* parameters for 800 time steps on 4 processors from a 64 processor run.[2] Clearly, there are two distinct type of spikes in the plots: big and small. There is also high correlation and similarity between the curves. Currently, we are examining whether the high cross-processor correlation is repeated in other applications. Regardless of the cross-processor correlation, existence of spikes is an evidence for a heavy tail component.

At this point, we do not know if the source of the observed variation in runtime between time steps is due to the application, or due to the system it runs on. For the purposes of designing a robust online tuning algorithm, the source of this variability is not important, but its properties (i.e. is it heavy tailed) is what matters.

Fig. 4 is the pdf of all 64 processors performance data. As we expect, the last three bars are not negligible, which indicates existence of a heavy tail component in the distribution.

One systematic way for checking heavy tails is to draw the log-log scale plot of 1-cdf, which is $P[X > x]$. For the heavy tail r.v., tail of the log-log plot should be approximately linear. Fig. 5 is the corresponding plot for the GS2 data and the last part of the graph approximately forms a line.

In order, to study characteristic of the small spikes in fig. 3,

[2]The cluster used has 64 nodes. Each node is equipped with dual Intel Xeon 2.66 GHz processors. Nodes are connected via Myrinet network, and used the PBS batch scheduler to admit at most one application per node at a time.

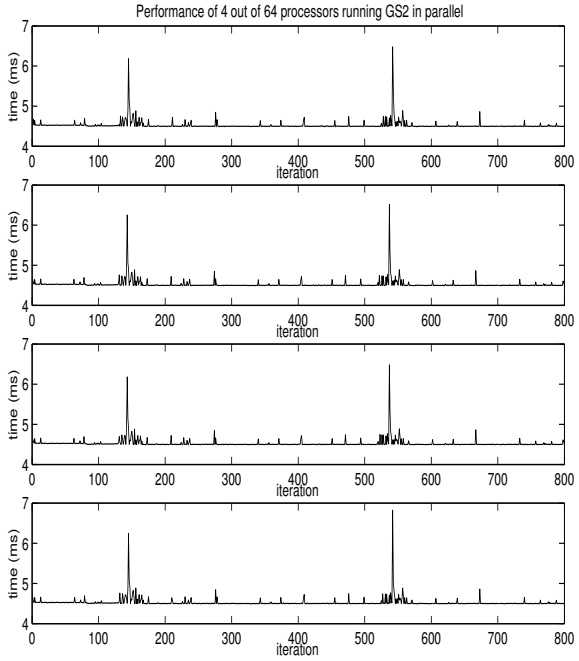


Figure 3: Running time for 800 iterations of the GS2 program on 4 out of 64 parallel processors

we truncate the GS2 data and remove all samples that are larger than 5. The pdf and 1-cdf plot for the truncated data is shown in fig. 6 and 7 respectively. Evidence for heavy tail component, which is due to the small spikes this time, is present in the plots.

In summary, the data presented in this section suggests that the performance variability is heavy tailed. As we show in the next section, the heavy tail assumption has profound impact on the characteristics and performance of the estimation and optimization algorithms.

5. VARIABILITY RESILIENT METHODS

In this section, we provide an optimization algorithm that is resilient to both heavy and non-heavy tail performance variability distributions. Before proceeding further, we have to review some properties of the random variables that we will use throughout this section.

Let $x_1, \cdots, x_k$ be a sequence of i.i.d. random variables with cdf function $F_X(x)$. Let $Q_X(x)$ be defined as below:

$$Q_X(x) = 1 - F_X(x) = P[X > x]. \tag{10}$$

Let $L_x^{(k)} = \min(x_1, \cdots, x_k)$. For $L_x^{(k)}$, we have:

$$\begin{aligned} Q_{L_x^{(k)}}(l) &= P\left[L_x^{(k)} > l\right] \\ &= P\left[\min(x_1, \cdots, x_k) > l\right] \\ &= \left[Q_X(l)\right]^k. \end{aligned} \tag{11}$$

5.1 Multiple Sampling and Minimization Operator

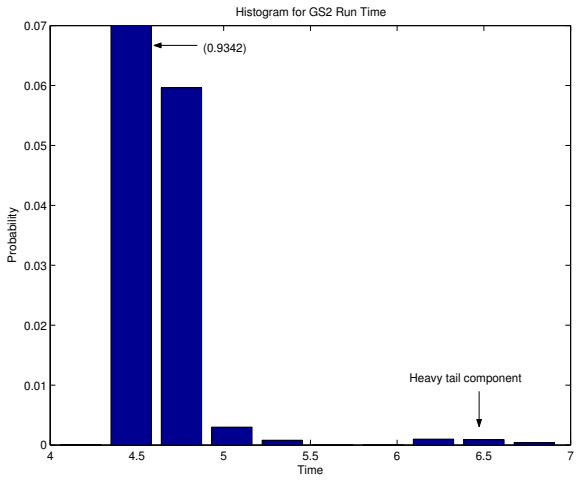


Figure 4: pdf of the GS2 data

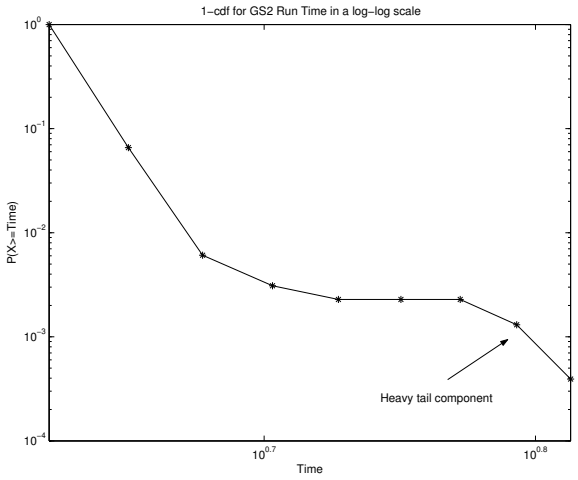


Figure 5: 1-cdf of the GS2 data

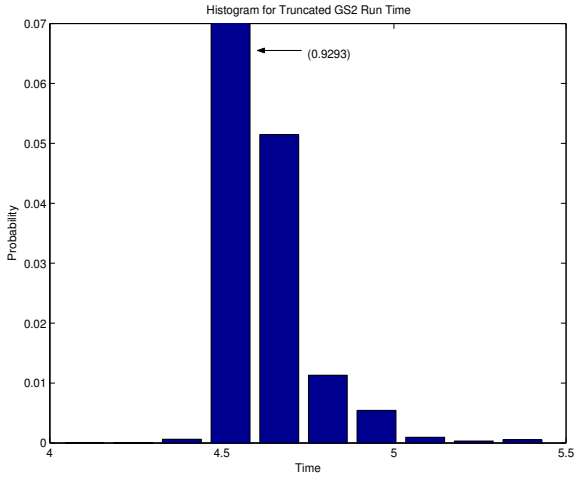


Figure 6: pdf of the truncated GS2 data

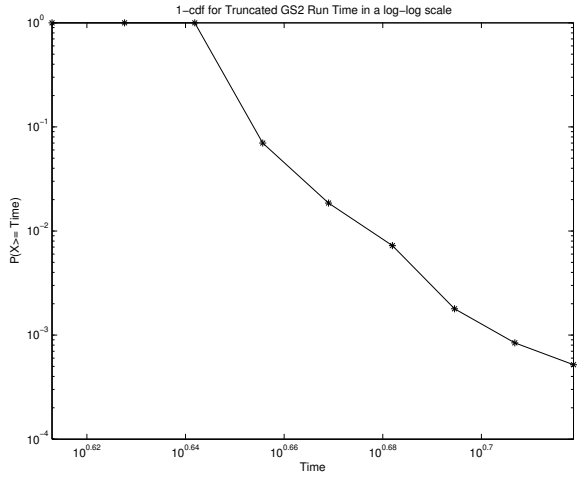


Figure 7: 1-cdf of the truncated GS2 data

The conventional method for dealing with stochastic variability in the direct search methods is to use multiple samples to estimate the function value at each desired point [19]. If we know the performance variability distribution, we can design an optimal estimator. However, in reality such information is not available, and we have to rely on more general estimators.

A natural choice for estimator is an *average* operator. If we take the average of multiple performance measurements, y in (5), one may *expect* that the average converges to the expected value of y. However, this is only true if y has finite variance. Recall that a heavy tail random variable does not have finite variance; therefore if y contains a heavy tail component we can not rely on the average operator.

Results in the previous section for GS2 data as well as data presented in [9, 15] suggest that performance variability distribution is heavy tailed. Consequently, we propose to use the $\min(\cdot)$ operator.

Let y_k be the kth sample for $f(v)$. From (5), we have:

$$y_k(v) = f(v) + n_k(v). \tag{12}$$

The min value is:

$$\begin{aligned} L_y^{(K)}(v) &= \min(y_1(v), \cdots, y_K(v)) \\ &= f(v) + \min(n_1(v), \cdots, n_K(v)). \end{aligned} \tag{13}$$

Let $n_{\min}(v)$ be the smallest value for $n(v)$ with non-zero probability. It is easy to show that for any $\varepsilon > 0$,

$$P[L_y^{(K)}(v) > f(v) + n_{\min}(v) + \varepsilon] \to 0 \text{ as } K \to \infty. \tag{14}$$

Note that $n_{\min}(v)$ is a deterministic function of $f(v)$. Therefore, if we can express $n_{\min}(v)$ as a function of $f(v)$, for sufficiently large k, $L_y^k(v)$ can be approximated as a function of $f(v)$,

$$L_y^{(K)}(v) \simeq f(v) + n_{\min}(v) = G\left(f(v)\right). \tag{15}$$

Conversely, after computing $L_y^K(v)$, we can use the inverse function $G^{-1}\left(L_y^K(v)\right)$ to estimate $f(v)$.

In reality, we do not have $G(\cdot)$, but on the other hand, we do not need to estimate $f(v)$ either. For the optimization algorithm, we only need to order the performance of the two alternative configurations. To that end, it is reasonable to assume that $n_{\min}(v)$ is an increasing function of $f(v)$. In that case, if we want to compare performance of two alternative parameter sets v_1 and v_2 we can use the following property:

$$\begin{aligned} &f(v_1) \textit{ is greater(less) than } f(v_2), \textit{ iff } f(v_1) + \\ &n_{\min}(v_1) \textit{ is greater(less) than } f(v_2) + n_{\min}(v_2). \end{aligned}$$

In other words, for comparing performance of two alternative parameter configurations v_1 and v_2 in the PRO algorithm, if we take enough samples so that the $\min(\cdot)$ operator converges, it is sufficient to compare $L_y^{(K)}(v_1)$ and $L_y^{(K)}(v_2)$.

As an example, let's assume that $n(v)$ has Pareto distribution and the system idle throughput is ρ. For the Pareto distribution we have,

$$E(n(v)) = \alpha\beta/(\alpha - 1) \tag{16}$$

From (7), (16):

$$\beta = \frac{(\alpha - 1)\rho}{(1 - \rho)\alpha} f(v) \tag{17}$$

Note that $n_{\min} = \beta$, which from (17) is a linear function of $f(v)$. Next, we compute the 1-cdf function for each sample $y = f(v) + n$, using relation (9) for Pareto distribution:

$$P[y > z] = P[n > z - f(v)] = \left(\frac{\beta}{z - f(v)}\right)^{\alpha}, \tag{18}$$

and using relation (11) for the minimum of K samples $L_y^{(K)}(v)$:

$$P[L_y^{(K)}(v) > z] = P[n > z - f(v)]^K = \left(\frac{\beta}{z - f(v)}\right)^{K\alpha}. \tag{19}$$

Interestingly enough the minimum of K samples have Pareto distribution with parameter $K\alpha$. Therefore, even when $0 < \alpha < 1$ and samples have infinite mean and variance, for $K > \alpha^{-1}$, the minimum has finite mean and variance and is not heavy tailed. This is a very interesting and appealing characteristic, which guarantees convergence of the min operator, even when samples are very unpredictable and have infinite mean and variance. The average operator does not have this property and that is the main reason for its failure. Moreover,

$$P[L_y^{(K)}(v) > f(v) + n_{\min}(v) + \varepsilon] = \left(\frac{\beta}{\beta + \varepsilon}\right)^{K\alpha}, \tag{20}$$

which satisfies (14).

5.2 Multiple-Sample Parallel Algorithms

In this section, we introduce a modification to the PRO algorithm that is more resilient to the performance variability. The modification is based on the previous section. Basically, instead of evaluating $f(v)$ only once, we evaluate it K times, where K is a fixed pre-determined integer. Then, we use the

minimum of K evaluation instead of $f(v)$ in the PRO algorithm. Basically, in Algorithm 2, we are replacing $f(\cdot)$ with $L_y^{(K)}(\cdot)$.

The main question is how to set the parameter K. In the PRO algorithm, we do not need to estimate the function value exactly, but we need to order function values at two alternative points v_1 and v_2 with minimal error. Suppose that the parameter space is discrete and finite. Define,

$$\Delta(v_1, v_2) = |f(v_1) + n_{\min}(v_1) - f(v_2) - n_{\min}(v_2)| . \quad (21)$$

Let $\lambda > 0$ be small enough so that if $\Delta(v_1, v_2) > 0$, then $\Delta(v_1, v_2) > \lambda$ for all admissible v_1 and v_2. Intuitively, λ is less than the least possible non-zero performance difference between two points.

From (14), we know that for any $\epsilon > 0$ there exists $K_0 > 0$ such that

$$P[L_y^{(K_0)}(v) > f(v) + n_{\min}(v) + \lambda] < \epsilon. \quad (22)$$

If we know λ, we can start with a desirable error probability $\epsilon > 0$, and compute sufficient number of samples K_0. In practice, it is not easy to find a fixed value for K. Currently, we are working on optimization algorithms that update K adaptively. In a parallel setup, depending on the number of parallel processors and tunable parameters, we might be able to calculate multiple samples in parallel without any additional time burden.[3] For instance, for the GS2, there are three tunable parameters. Therefore, we need to evaluate the performance at 6 different points for each time step. If there are 64 parallel processors running GS2 concurrently, we can set $K = 10$ with no additional cost.

6. SIMULATION RESULTS

In this section, we present simulation results for the PRO algorithm. First, we explore the effect of initial simplex size and shape on the performance. In the second part, we study performance of the modified PRO with multiple samples for different levels of idle throughput. To conduct a controlled study, we used a data base that contains the performance of the GS2 application for different parameter values, and simulated various optimization algorithms. There are 3 tunable parameters, however the data base does not contain all possible combinations. If a point is not in the data base, we use weighted average of its closest neighbors performance values to estimate its performance. Fig. 8 shows performance of the GS2 data base when one of the parameters is fixed. Clearly, the optimization surface is not smooth and contains multiple local minimums.

6.1 Initial Simplex

In this section, we study how the initial simplex size and shape affects the performance. We consider two alternative shapes for the simplex. The first simplex is a minimal simplex with $N + 1$ vertices. The first N vertices are $\{\Pi(c + b_i.e_i), \quad i = 1, \cdots, N\}$, where c is the center of the admissible region and $b_i = \frac{r(u(i)-l(i))}{2}$. Recall that $u(i)$ and $l(i)$ are upper and lower limits for the ith tunable parameter, and r, the initial simplex relative size, is a variable that specifies the simplex size. The second simplex has $2N$ vertices $\{\Pi(c \pm b_i.e_i), \quad i = 1, \cdots, N\}$.

[3]For the analysis to be valid, we would need independence of the variability on multiple processors during a single time step.

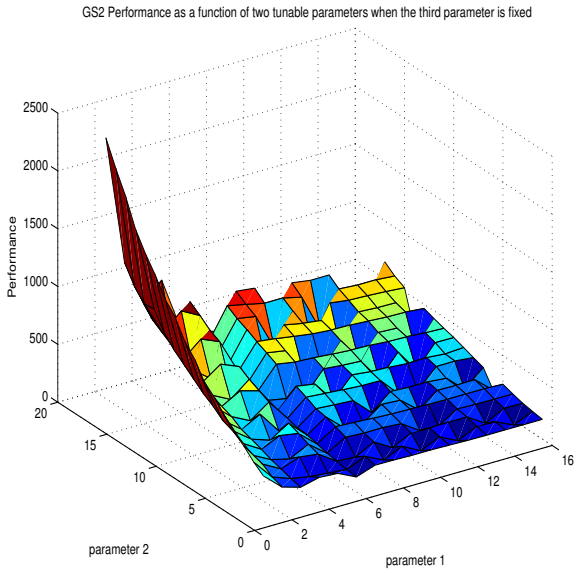


Figure 8: GS2 performance plot as a function of two tunable parameters, when the third parameter is fixed.

Performance for alternative simplex shapes and with different values of r is plotted in fig. 9. The $2N$ vertex simplex, clearly outperforms the $N + 1$ vertex simplex. Initial size of the simplex also has considerable effect on the performance of the algorithm.

Next, we focus on the $2N$ vertex simplex and study how its performance varies with r. We are not trying to find the optimum value for r here, since it is obviously problem dependent. Our intention is to come up with a general set of rules for setting up r. From the plot, we can infer that neither small nor large size initial simplexes likely perform well.

Regarding, large initial simplexes, this result is consistent with the results reported in [3], which concluded that extreme values for parameters often do not perform very well. The small values of r, when the simplex points are very close to each other, it may cause two problems. First, if there is a local minimum close to the center, the algorithm may get stuck in that neighborhood and not be able to explore other areas. Second, even if there is no local minimum close to the center, the simplex should go through several expansions, which are very costly in terms of performance, before finding the minimum. Based on these observations we use a $2N$ vertex simplex with $r = 0.2$ in the rest of simulations.

6.2 Multiple Sampling and Performance Variability

In this section, we study performance of the modified PRO with multiple samples for different idle throughput values.

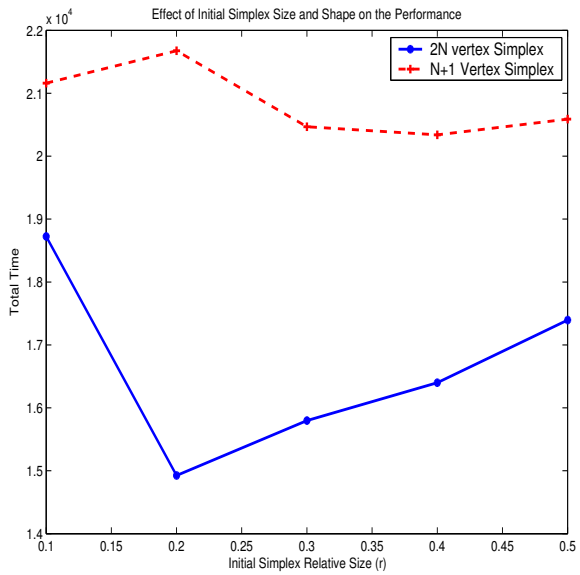


Figure 9: Average Normalized Total Time vs. Number of Samples for Different Idle Throughput Values

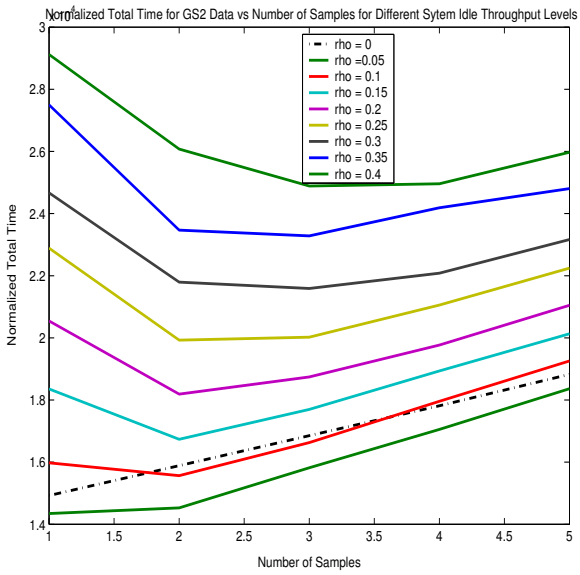


Figure 10: Average Normalized Total Time vs. Number of Samples for Different Idle Throughput Values

Performance variability, $n(v)$, is modelled as an i.i.d. Pareto distribution. The appropriate value for the Pareto distribution parameter α depends on the application and system, but for now we have set it to 1.7. The performance variability is, therefore, heavy tailed with finite mean and infinite variance.

The value of β is derived from relation (17). We assume that multiple samples for a single point are taken in subsequent time steps. Hence, we do not take advantage of multiple parallel sampling. Obviously, this assumption may not be true, but will create a worst case scenario for the multiple sampling case. In each experiment, we calculate $Total\_Time(100)$, which is the total time to run 100 time steps. Each configuration is specified by two parameters: idle throughput and number of samples. The number of samples range is from 1 to 5, and the idle throughput go from 0 to 0.4 in 0.05 steps. For each configuration, we run 2,000 simulations and compute the average $Total\_Time$.

From relation (6), the average finish time of the system increases by a factor of $1/(1-\rho)$; therefore, average $Total\_Time$ for different idle throughput values are not directly comparable. In order to make the $Total\_Time$ values comparable, we use $Normalized\_Total\_Time$ (NTT), where:

$$NTT = (1 - \rho)Total\_Time. \tag{23}$$

Simulation results are summarized in Fig. 10. First of all, notice that NTT for $\rho = 0$ increases linearly with number of samples, which is the expected result. Since there is no performance variation in this case, multiple samples of the same point are equal and redundant, and performance does not improve by multiple samples. However, since samples are taken in subsequent time steps, we are increasing the number of time steps, which results in linear increase of Total Time.

In other plots, there is an internal minimum point, which is the optimal number of samples for the corresponding idle throughput value. As we expect, the optimal number of samples increases as the idle throughput, and consequently performance variation, increases.

In general, the system performance decreases as the Performance Variation increases, with a very interesting exception. The $\rho = 0.05$ curve is below the $\rho = 0$ curve, which is counter-intuitive. This is possibly due to the complexity of the optimization surface. The optimization function has many local minimums, and therefore PRO may often trap in a local minimum basin of attraction and fail to find better solutions. A small value of the ρ can work as a controlled noise level that helps the system get out of low performance local minimums. This is similar to the concept used in the design of randomized optimization algorithms such as Simulated Annealing. So oddly enough, a little bit of performance variability can actually improve the overall performance when using on-line tuning.

7. RELATED WORK

Our prior work on Active Harmony [3, 18] looked at using historic data to infer the objective function and to demonstrate that the ideas could really improve an application performance. Our main contribution, in this paper, is to consider parallel optimization algorithms, study the impact of performance variations on the optimization algorithm, and to provide simple modifications to make the algorithm resilient to these variations.

Several other projects have looked at runtime tuning of applications. The Autopilot project [16, 17] allows applications to be adapted in an automated way. The AppLes project [2] and the Odyssey project [14] focus on resource awareness at the application level. In those systems, appli-

cations are informed of resource changes and provided with a list of available resource sets. The ATLAS [20] project has developed automatically tuned linear algebra libraries, their approach is an off-line tuning for a specific platform, rather than the Harmony approach of on-line tuning. The Nimrod/O project [1] tries to reduce the search space for engineering design. It applies multiple tuning algorithms including Simplex, P-BFGS, Divide and Conquer, and Simulated Annealing.

Characteristics and shortcomings of the Simplex algorithm are mentioned in several papers. Lagarias et al. [10] prove convergence for all 1-dimensional strictly convex functions. However, even for the 2-dimensional case, McKinnon [12] provides a family of strictly convex functions and a set of initial conditions for which the Simplex algorithm converges to a non-minimum. He also reports that the algorithm is not stable against small numerical perturbations caused by rounding errors.

Structure and convergence of the rank ordering algorithms are discussed in [5, 11]. Kolda, et al. [7] is a comprehensive review of the direct search algorithms.

Performance variation of parallel architectures are discussed in [9, 15]. However, their objective is to provide a systematic mechanism to find causes of variations, and then to fix them. Our approach is different; we model variations as a stochastic process and develop robust algorithms that function properly, even when application performance is variable.

8. CONCLUSION

We presented a Parallel Rank Ordering algorithm for tuning of application parameters. The algorithm belongs to a class of direct search algorithms known as GSS methods that are proved to have more reliable and predictable performance compared to the Simplex algorithm [7]. We discussed the impact of the performance variability on the behavior of the optimization algorithm. We presented preliminary results that indicate the performance variability of applications on clusters is heavy tailed. We explained that presence of a heavy tail component has a profound impact on the design of optimization algorithms. In particular, we showed that averaging multiple performance measurement will not necessarily give a better estimate of performance. As an alternative, we proposed to take the minimum of multiple performance measurements and showed that this is an effective way for performance estimation. We then, employed the minimization operator in the optimization algorithm to improve its performance. Finally, we simulated the algorithm and used data from a scientific application (GS2) with different parameter values, in order to study the algorithm characteristics and behavior.

9. ACKNOWLEDGMENT

This work was supported in part by NSF award EIA-0080206 and DOE Grants DE-CFC02-01ER25489, DE-FG02-01ER25510.

10. REFERENCES

[1] Abramson, D., et al., "An Automatic Design Optimization Tool and its Application to Computational Fluid Dynamics," *Proc. of SC'01,* Nov. 2001.

[2] Berman, F., R. Wolski, " Scheduling from the perspective of the application," *Proc. of IEEE Int. Symp. on HPDC,* 6-9, Aug. 1996.

[3] Chung, I.-H., J.K. Hollingsworth, "Using Information from Prior Runs to Improve Automated Tuning Systems," *Proc. of SC'04,* Nov. 2004.

[4] Crovella, M., and A. Bestavros., "Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes," *IEEE/ACM Trans. on Networking,* Dec. 1997.

[5] Dennis, J.E., V. Torczon, "Direct Search Methods on Parallel Machines," *SIAM J. on Opt.,* 448-474, vol.1, 1991.

[6] Dorland, W., F. Jenko, M. Kotschenreuther, B.N. Rogers, "Electron Temperature Gradient Turbulence," *Physical Review Letters,* 5579-5582, vol.85, No.26, 2000.

[7] Kolda, T.G., R.M. Lewis, V. Torczon, "Optimization by Direct Search: New Perspectives on Some Classical and Modern Methods," *SIAM Review,* 385-482, vol.45, No.3, 2003.

[8] Kotschenreuther, M., G. Rewoldt, W.M. Tang, " Comparison of Initial Value and Eigenvalue Codes for Kinetic Toroidal Plasma Instabilities," *Computer Physics Communications,* 128-140, vol.88, 1995.

[9] Kramer, W.T.C., C. Ryan, "Performance variability of highly parallel architectures," *Lawrence Berkeley National Laboratory,* Paper LBNL-52291. http://repositories.cdlib.org/lbnl/LBNL-52291, May, 2003.

[10] Lagarias, J.C., J.A. Reeds, M.H. Wright, P.E. Wright, "Convergence Properties of the Nelder-Mead Simplex Method in Low Dimensions," *SIAM Journal on Optimization,* 112-147, vol.9, No.1, 1998.

[11] Lewis, R.M., V. Torczon, "Rank Ordering and Positive Bases in Pattern Search Algorithms," *Tech. Rep., Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, VA,* 96-71, 1996.

[12] McKinnon, K.I.M, "Convergence of the Nelder-Mead Simplex Method to a Nonstationary Point," *SIAM Journal on Optimization,* 148-158, vol.9, 1998.

[13] Nelder, J.A., R. Mead, "A Simplex Method for Function Minimization," *Computer Journal,* 308-313, vol.7, 1965.

[14] Noble, B.D., et al., "Agile Application-Aware Adaptation for Mobility," *ACM Symp. Operating Systems Principals,* 276-287, 1997.

[15] Petrini, F., D.J. Kerbyson, S. Pakin, "The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q," *Proceedings of the ACM/IEEE SC2003,* Nov., 2003.

[16] Ribler, R.L., J.S. Vetter, H. Simitci, and D.A. Reed,"Autopilot: Adaptive Control of Distributed Applications," *Proc. of the IEEE Int. Symp. on HPDC,*p.172, 1998.

[17] Ribler, R.L., H. Simitci, and D.A. Reed,"The Autopilot Performance-Directed Adaptive Control System," *Future Generation Computer Systems,*175-187, vol.18, no.1, 2001.

[18] Tapus, C., I.-H. Chung, J.K. Hollingsworth,"Active Harmony: Towards Automated Performance Tuning," *ACM Transactions on Computer Systems,* 319-352, Nov., 2003.

[19] Trosset, M.W.,"On the Use of Direct Search Methods for Stochastic Optimization," *Technical Report 00-20, Department of Computational & Applied Mathematics, Rice University,* June, 2000.

[20] Whaley, R.C. and J.J. Dongarra, "Automatically tuned linear algebra software (ATLAS)," *Proc. of SC'98,* 1-27, 1998.