ANGEL: A Hierarchical Approach to Multi-Objective Online Auto-Tuning

Ray S. Chen
Department of Computer Science
University of Maryland
College Park, MD USA
rchen@cs.umd.edu

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

ABSTRACT

The majority of auto-tuning research in HPC to date has focused on improving the performance of a single objective function. More recently, interest has grown in optimizing multiple objective functions simultaneously, for example balancing the trade-off between execution time and energy consumption. Evolutionary algorithms for multi-objective optimization attempt to quickly and accurately discover the set of Pareto-optimal solutions, or the Pareto frontier. However, a single solution (as opposed to a set) is desirable for online end-to-end auto-tuning. As an alternative, we use the hierarchical method based on prioritizing the objective functions and iteratively optimizing them in isolation to efficiently produce a single Pareto-optimal result. This approach has several advantages over evolutionary or even scalarization techniques for auto-tuning HPC problems. In this paper, we introduce the Automated Navigation Given Enumerated Leeways (ANGEL) auto-tuning method. We demonstrate the quality of ANGEL using a multi-objective benchmark test suite. We also demonstrate its utility by approximating various sections of a Pareto front from a real-world proxy application. ANGEL successfully navigates trade-offs that range from a 15% reduction in kernel run-time to a 3% savings in energy use.

1. INTRODUCTION

Auto-tuning has established itself as an important tool in High Performance Computing (HPC) and the endeavor towards exascale computation. Empirical methods such as auto-tuning have been used in place of complex analysis to optimize everything from linear algebra libraries to parallel multicore stencil computations. The motivation is easy to understand; many optimization problems are too complicated or difficult to model. Consider optimizing the compilation of scientific application code. The optimal values for compiler transformations such as loop unrolling and tiling factors are highly dependent on the execution environment. Simple empirical trials are a welcome alternative to the com-

plexity of manually reasoning about hardware architecture artifacts (CPU/GPU count, cache hierarchy levels/sizes/associativity, etc.).

HPC optimization has traditionally focused on minimizing program execution time. Auto-tuning a single objective such as run-time is a well understood problem. A host of optimization algorithms were developed decades ago, several of which are naturally suited for auto-tuning. However, tuning for a single objective may soon be insufficient. The exascale computing initiative introduces several hurdles which must be considered alongside execution time. First and foremost is the 20MW power goal that introduces energy efficiency as an additional objective. Other considerations might include improving resilience and reliability in the face of billion-way parallelism.

Transitioning auto-tuning to handle multiple objectives is non-trivial. In the single-objective case, an algorithm can easily determine if an outcome has improved (e.g., “More money is better than less money.”). In the multi-objective case, determining improvement becomes more difficult (e.g., “Is it better to have more money, or more free time?”). Trade-offs between conflicting objectives produce a set of equally valid solutions. Selecting a single solution from this set is typically done by a human because the decision is subjective.

For exascale systems, having humans make these choices will not be practical. To operate in the complex and dynamic environment of likely exascale architectures, auto-tuning must become a fully online activity that is integrated into the runtime system. Moreover, it must coordinate all auto-tuning tasks simultaneously. The days of individual auto-tuners running in isolation are numbered.

In this paper, we provide first steps towards runtime system coordinated auto-tuning. We show that multi-objective HPC problems may be auto-tuned at runtime given two simple inputs: a priority and an allowable distance from optimal for each objective (e.g., “I am willing to forfeit up to 10% of my money to gain as much free time as possible.”).

2. RELATED WORK

To aid our discussion of multi-objective problems, we first provide a brief overview of Pareto optimality. We use a minimization problem with two objectives to demonstrate. Consider two points (α and β) in the objective result space. If $\alpha \Rightarrow (3, 3)$ and $\beta \Rightarrow (5, 5)$, then α is said to dominate β because α is better than β in all objective dimensions. In this case, β can freely be discarded. Now consider a third point γ . If $\gamma \Rightarrow (2, 4)$, then γ and α do not dominate one

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ROSS '15, June 16, 2015, Portland, Oregon, USA

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ACM 978-1-4503-3606-2/15/06 ...\$15.00.

<http://dx.doi.org/10.1145/2768405.2768409>

(a) Progress of an evolutionary algorithm as evaluations increase. (b) Progress of ANGEL compared to an evolutionary algorithm.

Figure 1: On the left, we show an evolutionary algorithm (ParEGO) against the multi-objective test function OKA1. As we increase the number of evaluations, the resulting Pareto frontier gets closer and closer to truth (determined by hand). On the right, we see that ANGEL uses its evaluations to drill down towards a single point on the true Pareto frontier. For comparison, we overlay the Pareto frontier as discovered by ParEGO using the same number of evaluations as ANGEL.

another, and the choice between them becomes less clear. These solutions represent trade-offs between objective functions, where improving one objective necessarily results in the degradation of another. The set of all non-dominated points is called the Pareto-optimal set, or the Pareto frontier.

A straightforward approach to multi-objective optimization is scalarization, described in several surveys from varying disciplines [1, 6, 11]. Scalarization is a method of reducing multi-dimensional objective spaces down to a single dimension. This enables auto-tuners to use tried-and-true single-objective optimization algorithms. For example, the weighted-sum approach multiplies each objective by a weight factor before summing them all to form a single value. The weights represent the relative importance of each objective and must be provided before the search may begin. However, choosing appropriate weights requires knowledge of the objective's value ranges. Without normalization, the value range for each objective may differ from one another by orders of magnitude. This makes it extremely difficult and impractical to correctly articulate user preferences using weight factors alone.

As an alternative, evolutionary algorithms can be used to solve the multi-objective optimization problem directly. They work by discovering the Pareto frontier and producing a Pareto-optimal set of solutions. This is demonstrated in Fig. 1a which shows the progress of an evolutionary algorithm. Evaluations are marked as points in the result space, and a Pareto frontier line is drawn after 32, 64, 128, and 256 evaluations. As the number of evaluations increases, the calculated Pareto frontier approaches the true Pareto frontier.

However, evolutionary algorithms are inefficient in terms of function evaluations. It is common for such algorithms to use thousands of evaluations or more to produce their results. Auto-tuning HPC applications often involve objective

functions that are expensive to evaluate (i.e., queuing, running, and timing a job for every evaluation). To be feasible in our domain, the number of evaluations to converge upon a solution must be reduced by several orders of magnitude.

Recent advances in evolutionary algorithms address the issue of function evaluation efficiency. Knowles [5] targets problems whose objective functions are prohibitively expensive to evaluate. Using landscape approximation, his algorithm (ParEGO) is able to achieve competitive results using only 250 function evaluations. Zuluaga et al. [15] improves upon Knowles' results by using active learning to achieve a 33% reduction in function evaluations (although it is unclear if evaluations for training runs were considered).

Still, the goal of these algorithms is to discover the Pareto frontier; they do not aid in the final selection. Simply producing a Pareto-optimal set of solutions is insufficient for online end-to-end auto-tuning. Without further instruction, there is no way for an auto-tuning system to distinguish between these solutions. A human is required to select the best of the bunch. Any auto-tuning system that requires human participation during the search can hardly be considered automatic tuning.

In the HPC domain, Jordan et al. [4] focuses on HPC kernels, automatically optimizing them for execution speed and parallel efficiency. He demonstrates that a differential evolution algorithm may be used to efficiently reduce the search space. Roughly 1,000 evaluations were needed to approximate the Pareto set for each experiment involving three decision parameters and two objective functions. The intended environment for Jordan's framework is an optimizing compiler where an offline multi-objective search is conducted to find the Pareto frontier. The set of solutions are compiled into a multi-versioned binary so that all are available during a production run of the kernel. A method for selecting from the available versions during a production run is not addressed.

In contrast to Jordan, our work is geared towards an on-line auto-tuning environment. This has two implications. Firstly, we aim to produce gains within one execution of the application. This requires the search to converge well before the application terminates. Otherwise, it becomes increasingly difficult for any performance improvements to make up for the overhead of the concurrent auto-tuning search. Secondly, there is no time to ask the user for his or her preferences a posteriori. The search must choose a solution from the Pareto-optimal set automatically.

3. THE ANGEL SEARCH METHOD

To avoid the “a posteriori” nature of evolutionary algorithms and the “a priori” normalization needed for scalarization, we adopt the hierarchical optimization method originally described by Waltz [14]. It is similar to lexicographical or multilevel optimization [7], where the objectives are first given a priority ranking. Then, a single-objective algorithm is iteratively used to optimize each objective in priority order. Constraints discovered from upper levels of the hierarchical search are passed down and inform lower levels. Values are normalized on the fly between iterations, which allows for the comparison of objectives with wildly different value ranges.

More simply, hierarchical optimization allows users to articulate preferences using two values per objective; a leeway percentage and a priority ranking. The leeway percentage value represents the objective’s acceptable margin of freedom. For example, valid solutions must stay within 10% of the best-known value for a given objective. The priority ranking represents the objective’s relative importance. In case of conflict, the leeways of higher priority objectives are honored before those with lower priority.

With these properties in mind, we propose the Automated Navigation Given Enumerated Leeways (ANGEL) auto-tuning method. ANGEL is designed to:

- Eliminate the need for human intervention during the search phase. A single solution will be returned after user preferences are articulated.
- Allow optimization without knowing objective value ranges a priori.
- Require relatively few function evaluations for convergence. Ideally, fewer than 100 for practical usage in the HPC domain (two or three objective functions).
- Minimize the total sum of evaluated objective function values to aid online auto-tuning (e.g. time and power spent on the optimization search.)
- Require low computational complexity to reduce run-time overhead.
- Perform at least as well as evolutionary algorithms in terms of converged solution quality.

Instead of approximating an entire Pareto frontier, user preferences guide the search towards the “best” optimal solution. This frees the search algorithm to spend its empirical capital on finding the single user-preferred solution, rather than a set of optimal solutions.

This is visually demonstrated in Fig. 1b. We plot an ANGEL search through the result space of the OKA1 multi-objective test problem [9]. In this case, we allow ANGEL

a 55% leeway for objective #1, and prioritize objective #1 over objective #2. Guided by these user inputs, ANGEL drills directly towards a particular result on the Pareto frontier, finally converging on a solution very close to the true Pareto frontier. For comparison, Fig. 1b also shows a search by the evolutionary algorithm ParEGO when allotted the same number of function evaluations ANGEL required for convergence. ParEGO approximates the Pareto frontier well, but ultimately ends rather far from the Pareto-optimal solution 55% away from the best-known value for objective #1 (far left black circle). Put another way, ANGEL gets closer to the Pareto frontier more quickly by favoring depth over breadth.

3.1 Implementation

ANGEL operates by iteratively performing N sub-searches, where N is the number of objective functions. Each sub-search considers its objective in isolation, which enables ANGEL to rely on single-objective search strategies. The objectives are considered in descending order of priority and, between each sub-search, the leeway percentage is used to calculate a threshold for the prior objective. Solutions that violate threshold values are not considered in subsequent sub-searches.

A penalty function is used to enforce the threshold values. This specialized function ushers sub-searches towards objective values that do not violate thresholds of higher-priority objectives. The penalty function is detailed in Sect. 3.4.

The overall multi-objective search converges after performing a single-objective search on the lowest priority objective. This final sub-search considers all $N - 1$ calculated thresholds and effectively coerces the final single-objective sub-search to optimize N objectives at once.

3.2 Underlying Single-Objective Algorithm

Conceptually, any single-objective search algorithm may be used for ANGEL’s subordinate search. There are many high quality single-objective optimization algorithms from which to choose. For instance, the Nelder-Mead [8] algorithm was first published in 1965 and remains popular to this day. It uses a simplex to approximate the local gradient of an underlying objective function, and defines four basic simplex transformations that allow it to move towards a minimum value.

We use the Nelder-Mead downhill simplex algorithm for two reasons. Firstly, simplex-based methods generally require fewer function evaluations for convergence when compared to evolutionary or stochastic approaches. ANGEL requires its subordinate search to run multiple times, so converging quickly is of the utmost importance. Secondly, simplex-based methods are computationally quick and inexpensive. This enables ANGEL to be used for online auto-tuning scenarios like run-time code optimization which requires just-in-time compilation.

The convergence method for the underlying algorithm is also important. We use the method described by Parkinson and Hutchinson [10], where convergence is based on how far the simplex moves at each step. Once the simplex moves less than 3% of the total search space for three consecutive search steps, the sub-search is considered converged. This avoids the need to reason about values in the result space, which is impossible without a priori knowledge of the objective functions.

Figure 2: Induced search space of successive ANGEL search iterations. A threshold value is discovered at the end of each search phase, and applied during future search phases. Notice that ANGEL is not affected by the large difference between each objective function’s value range.

3.3 Leeway and Thresholds

ANGEL requires $N - 1$ leeway percentage values that represent an acceptable margin of freedom from the best-known value for each objective. During function evaluation, a minimum and maximum observed value is updated for each objective. In effect, this bootstraps a value range for each objective without relying on a priori knowledge. Upon sub-search convergence, the leeway percentage is applied to the value range to produce a threshold value.

More concretely, the threshold value is calculated according to the relationship

$$threshold_i = min_i + (max_i - min_i) * leeway_i,$$

where i is the current objective function. Once calculated, any configuration that produces a value for objective i greater than $threshold_i$ will be penalized.

3.4 Penalty Function

Our penalty function is used to guide a single-objective search to optimize several objectives at once. We achieve this by considering how badly a threshold is violated. If a threshold is barely violated, a relatively small penalty is applied to the current objective. Similarly, a large penalty is applied for large threshold violations. This produces a continuous penalty function that is better suited to simplex-based optimization algorithms.

Formally, each objective that exceeds its threshold increases the penalty by a basic unit defined as

$$base_i = global_max_i - global_min_i,$$

where $global_min_i$ is the minimum and $global_max_i$ is the maximum value ever observed for objective i . Note that this range may be different from the range used in the threshold relation (min, max) . The former is recorded after every function evaluation, and the latter is only updated during its particular sub-search iteration. This allows threshold values to remain constant after they are generated, which provides stability to subsequent phases of the greater search.

The formal definition of how badly a threshold has been violated is defined by

$$severity_i = \frac{obs_i - global_min_i}{global_max_i - global_min_i},$$

where obs_i is the observed value for objective i . The value of $severity_i$ is represented by a percentage similar to leeway values. A geometric term is added to maintain the priority ranking, which produces the the penalty factor

$$factor_i = 2^{N-i} + severity_i,$$

where N is the total number of objectives. The geometric sequence used in this term will enforce a priority hierarchy even after the final summation of violation penalties as defined by

$$penalty_k = \sum_{i=1}^{k-1} \begin{cases} 0, & \text{if } obs_i \leq threshold_i \\ factor_i * base_k, & \text{otherwise} \end{cases}$$

where k is the current sub-search iteration, also referred to as the ANGEL search phase.

Notice that, while $penalty_k$ is influenced by $k - 1$ prior objective functions, each objective function only contributes a

percentage value. The penalty is applied in terms of the k th objective function when multiplied by $base_k$. This eliminates the need for normalization of values between different objective functions.

3.5 Visual Example

As a concrete example, consider the multi-objective test function VLMOP3 [13]. It derives three objective functions from two input variables, as seen in the top row of Fig. 2.

Each objective function is the basis for an individual iteration (search phase) of ANGEL. As the search progresses, our penalty function alters the original objective surface when thresholds from prior objectives have been violated. The penalty function preserves the smoothness of the original objective (via the *severity_i* term), which reduces the chance that simplex-based search methods become stranded.

The smooth nature of our penalty function can clearly be seen in each ANGEL search phase. The concentric ripples from objective 1 can be seen in search phase 2 and 3, and the vertical trough from objective 2 in search phase 3.

4. SYNTHETIC EXPERIMENTS

We begin our evaluation of ANGEL with the suite of test functions described by Knowles in his work on ParEGO [5]. These functions were collected from test suites popular in the multi-objective optimization literature [2, 9, 13]. In brief, this suite includes functions with low solution density, multiple local Pareto fronts, disconnected objective result space, non-uniform density, or some combination of these traits. More simply, they are designed to contain Pareto frontiers that are challenging to discover, but not pathologically so.

4.1 Methodology

We compare ANGEL against two different algorithms: random and evolutionary. The evolutionary algorithm we chose for comparison is ParEGO [5]. Since reducing function evaluations is a primary goal for ParEGO, it represents the best that evolutionary algorithms have to offer for the criteria important to online auto-tuning. A random algorithm is also included for comparison to ensure our algorithm is behaving intelligently.

Quantifying Success

In order to quantitatively evaluate or compare these algorithms, we first define our method of selecting the target solution. Our goal is to see how well ANGEL does at providing a solution to the hierarchical optimization problem. In order to make this comparison for ParEGO and random we need to select single operating point rather than the Pareto-optimal set that each of these techniques normally supply as output.

For ParEGO and random, we first filter out solutions from the Pareto-optimal set that violate tolerance thresholds. Of the remaining solutions, we select the one with the best value of the lowest priority objective (since all higher priority ones are within the user supplied tolerance from optimal).

For our ground truth solution, we performed a near exhaustive sweep of each synthetic function’s parameter space.

Mitigating Randomness

Each of the tested algorithms contain (or are affected by) a random element. To mitigate the effects of random behav-

ior, we ran 1,000 trials of each algorithm and report mean values for our comparison metrics. The metrics used in this experiment are detailed in Sect. 4.2.

Since we use the Nelder-Mead downhill simplex method as our underlying single-objective algorithm, ANGEL is similarly sensitive to initial values for simplex size, position, and orientation. To circumvent this sensitivity, each trial randomly translates and rotates the initial simplex. Also, the initial simplex size may range between 25% to 45% of the overall search space.

Determining Convergence

Both the random and the evolutionary algorithm are designed run for a limited amount of time or function evaluations. They do not converge, per se. To provide a fair comparison, we limit these algorithms to the average number of function evaluations needed for ANGEL to converge.

In the following tests, the priority ranking is taken directly from the function definition (the first objective has the highest priority). A 25% leeway is used for the primary objective and a 35% leeway is used for the secondary objective.

4.2 Comparison Metrics

The algorithms are compared using two metrics: search quality and efficiency.

Search Quality

We define search quality as the Euclidean distance between the converged solution and the true preferred optimal. It can be thought of as the best solution found using a given algorithm. Stable (non-noisy) problems would continue to use this solution once the algorithm has completed or converged.

Search Efficiency

We define search efficiency as the average Euclidean distance (from the true preferred optimal) per evaluation during the search. It can be thought of as the balance that an algorithm strikes between exploration and exploitation.

This metric is especially important for online auto-tuning. As Tiwari et al. have noted, the losses incurred by evaluating several poor solutions can easily outweigh the benefits of discovering an excellent solution [12].

4.3 Results

The benchmark results for each algorithm are shown in Table 1. The best algorithm in terms of quality and efficiency are shown in bold. We also show the average number of evaluations needed for ANGEL to converge.

Comparison to Random

Random algorithms are notoriously difficult to beat in terms of search quality. It is then no small feat that ANGEL meets or exceeds the quality of a random search for two thirds of the test suite. In general, the quality of ANGEL deteriorates as the parameter space increases in dimension. This is a result of using the Nelder-Mead algorithm for ANGEL’s subordinate search, which is known to have trouble with higher dimensions [3].

Search efficiency is a particular problem for the random algorithm. With no way to control which solutions are tested, the random algorithm is at the mercy of functions with a large percentage of high-cost solutions. The result table

Table 1: Comparison of multi-objective algorithms on a synthetic test suite. Data averaged from 1,000 runs of each algorithm on each function. Distance measured by ℓ^2 norm in the result space. The best metric for each function is highlighted in bold.

Test Function	Dimensions		Average Evaluations	Search Quality (Average convergence distance from optimal.)			Search Efficiency (Average distance from optimal per evaluation.)		
				Random	ParEGO	ANGEL	Random	ParEGO	ANGEL
	In	Out	ANGEL						
KNO1	2	2	62	2.385	2.950	2.081	11.665	7.711	6.800
OKA1	2	2	53	1.605	2.783	0.833	4.294	5.124	2.444
OKA2	3	2	59	2.294	2.978	2.068	5.903	5.891	3.391
VLMOP2	2	2	56	0.119	0.055	0.377	0.627	0.637	0.326
VLMOP3	2	3	87	1.172	1.371	1.231	7.808	4.567	2.378
DTLZ1a	6	2	141	67.889	3.368	0.840	220.070	118.498	30.346
DTLZ2a	8	3	209	0.893	0.935	0.665	0.963	0.925	0.651
DTLZ4a	8	3	241	1.080	0.678	1.415	1.792	1.499	1.451
DTLZ7a	8	3	223	12.992	8.351	6.327	12.721	9.135	11.646

clearly illustrates this point; it manages to produce the worst search efficiency in eight out of nine tests.

Comparison to Evolutionary

In terms of search quality, ANGEL outperforms ParEGO in seven of the nine tests. This is because ParEGO focuses on approximating the entire Pareto frontier, favoring breadth over depth.

Search efficiency is also a problem for ParEGO when compared to ANGEL, as many high-cost solutions are evaluated. This can be seen visually in Fig. 1a where, even late in ParEGO’s search (evaluations 129 through 256), ParEGO evaluates solutions extremely far from the Pareto frontier.

Additionally, evolutionary algorithms require a far greater amount of computation time to execute. Case in point, ANGEL ran orders of magnitude faster than ParEGO in this experiment due to the relatively simple underlying Nelder-Mead algorithm. The longer calculation time used by evolutionary algorithms is acceptable for fields such as engineering design optimization. Their multi-objective optimization problems are temporally expensive, with individual function evaluations taking hours or days. However, lengthy computation delays between experiments becomes an issue in HPC contexts such as optimizing the loop nests of a scientific application.

5. REAL-WORLD EXPERIMENT

Finally, we investigate the utility of ANGEL against real-world data. Specifically, we consider a performance vs. energy usage trade-off on a GPU. We use the OpenACC implementation of Lawrence Livermore’s LULESH¹ as the basis of our experiment. The dominant kernel within this proxy application (`CalcHourglassControlForElems()`) accounts for almost 30% of execution time on the GPU. This kernel is the target for our optimization experiment.

This experiment demonstrates the utility of ANGEL’s leeway parameter. We perform a series of experiments that gradually change the leeway value and show the progression of selected solutions.

5.1 Methodology

Our experiment manipulates two input variables. The first is the vector length used to vectorize `CalcHourglass-`

¹<https://codesign.llnl.gov/lulesh.php>

`ControlForElems()`. This parameter is provided by OpenACC and affects the parallelism allocated to the loop iterations of this kernel. The valid input range for this parameter goes from 4 to 256 in strides of 4. The second input variable is the GPU clock frequency. It is provided by the GPU hardware itself and is manipulated by `nvidia-smi`, an application distributed with NVIDIA’s Management Library (NVML). There are five reasonable states for this parameter (758MHz, 705MHz, 666MHz, 640MHz, and 614MHz).

Our experiment then measures two output variables. The first is elapsed wall time, as reported by the LULESH application itself (`MPI_Wtime()` is used to collect the data). The second is average GPU energy consumption as reported by the NVML. Using the Performance API’s (PAPI) NVML component as an intermediary, the GPU’s power consumption was sampled at the rate of 50 Hz. The final energy values we collect are a product of the mean power and the total elapsed time.

We conducted an exhaustive search of the parameter space, executing five trials of each possible configuration. This reveals a natural trade-off between runtime and energy consumption. The Pareto front for this problem consists of eight points (listed in Table 2). To isolate the time taken by the kernel itself, we enable a lightweight profiler via the environment variable `PGI_ACC_TIME`. This reports that `CalcHourglassControlForElems()` takes 24.8 seconds to run on average, meaning our Pareto front ranges from a 15% reduction in kernel runtime to a 3% savings in energy use.

Table 2: OpenACC LULESH time/energy Pareto frontier.

Input Space		Objective Space	
Vector Length	GPU Clock Rate (MHz)	Walltime (s)	Energy (kJ)
4	640	141.09	10.634
4	666	138.38	10.806
4	705	135.16	10.843
8	705	135.04	10.866
24	705	135.03	10.993
28	705	131.70	11.403
12	758	131.05	11.445
20	758	130.94	11.529

5.2 Experimental Setup

Our test hardware is a heterogeneous compute node (CPU:

Figure 3: These plots show the ANGEL search for three different values of the leeway parameter. The red +’s show the Pareto frontier, while the blue circles show the configurations tested by ANGEL.

Intel Xeon E5-2670, GPU: NVIDIA Tesla K20c) with 32 gigabytes of RAM. The software stack consists of RedHat Enterprise Linux (release 6.4, kernel version 2.6.32-358.23.2), running LULESH (GPU model, version 2.0) compiled using Portland Group’s PGI Accelerator (version 14.1), configured to use CUDA (version 5.5) and NVIDIA’s kernel driver (version 319.82).

To configure ANGEL, we prioritize the runtime and energy objectives as primary and secondary, respectively. Then, a series of searches are performed with runtime leeways of 0.1%, 1.0%, and 3.0%. Each search starts with the same initial simplex.

5.3 Results

The search space for this problem is much smaller than our synthetic tests with only 320 possible configurations. As our experiment will show, ANGEL still efficiently navigates the available space.

The plots in Fig. 3 illustrate how ANGEL investigates the result space. They show how ANGEL reacts when given different leeways for the same optimization problem.

We begin with the most restrictive 0.1% leeway (Fig. 3a). Execution time is plotted along the x-axis, and energy consumption is plotted along the y-axis. The vertical dashed line represents the threshold value discovered during the first iteration (search phase) of ANGEL. The empty circles represent solutions actually evaluated by ANGEL. Finally, the solid circle represents the solution ANGEL converges upon. In this case, effectively no leeway is given; only one Pareto-optimal solution exists to the left of the threshold line. This point is discovered by ANGEL and is appropriately proposed as the optimal point.

Using a 1.0% leeway (Fig. 3b), the threshold line moves to the right, consequently allowing two additional Pareto-optimal solutions into the valid region. During the search, ANGEL investigates several solutions beyond the threshold, each time being penalized and ushered back towards the left. This allows ANGEL to evaluate more solutions in the valid region and converge on the solution that minimizes energy use without violating the threshold. Again, ANGEL discovers and proposes the appropriate optimal point.

Finally, using a 3.0% leeway (Fig. 3c), ANGEL has been authorized to seek even greater energy savings. The careful observer will notice that ANGEL does not select the

Table 3: Best solutions for increasing ANGEL leeway percentages.

Leeway	Best Time	Best Energy	Distance from Optimal
0.1%	130.94	11.530	0.00
1.0%	131.70	11.403	0.00
3.0%	135.04	10.866	0.12

preferred optimal solution. One additional Pareto frontier node exists that is closer to the threshold line. In this case, ANGEL’s simplex converges before it is able to discover a better solution. This is always a possibility when using Nelder-Mead or any other heuristic-based search, as discovery of global optimal is not guaranteed. Instead, ANGEL selects the most energy efficient solution (it was aware of) that did not violate the threshold.

This is an excellent reminder that the underlying search algorithm is a configurable aspect of ANGEL. We selected a simplex-based method to favor speed and efficiency. If thoroughness becomes more desirable than search performance, ANGEL can easily be adapted to provide such a trait.

5.4 Discussion

This experiment shows that ANGEL works effectively not only on synthetic optimization problems, but also on actual performance curves in the HPC domain. A system-wide runtime study would be an even better demonstration of ANGEL’s suitability for managing HPC performance/power tradeoffs. For instance, an online auto-tuner like Active Harmony² (either linked with the target application or launched externally via the system OS) could monitor the target application, adjusting system power “knobs” as necessary.

Such an experiment requires two things. First, a system capable of obtaining accurate power measurements. Such power measurements must be available for all components of the system, and attributable to a single job (no cross-job interference). Second, an application with tunable parameters that impact power and performance.

Currently, such an application and system combination is not readily available. The industry has only recently begun to provide the facilities necessary for measuring the power of

²<http://activeharmony.org>

processing and memory units. Similar facilities for storage units and interconnection networks are still largely unavailable. Job attribution has yet to be addressed.

Once available, such resources could open several avenues of research for runtime systems and ANGEL. For example, it becomes possible to investigate dynamic power capping. Most power capping research to date limit consumption by a single static value. Runtime system coordination may be able to relax these limits, temporarily allowing certain compute nodes to exceed static power caps, so long as overall power draw remains within system-level constraints.

6. CONCLUSIONS

In this paper, we describe and evaluate ANGEL, a new hierarchical optimization algorithm tailored to the needs of online multi-objective auto-tuning in the HPC domain. Our method requires minimal input from users, makes no assumptions regarding objective values, and recommends a single solution as opposed to a set of equivalent solutions. This eliminates the need for a subjective final selection, which normally requires human intervention, bringing the field one step closer to a fully-automated reality.

We evaluate ANGEL using a rigorous synthetic test suite. We also demonstrate ANGEL's utility on the energy and performance curves of an HPC proxy application, in lieu of a fully power-sensitive system and application.

Many sectors of the HPC community could benefit from an automated technique like ANGEL. Trade-off scenarios will be common as we move towards exascale computing. Energy, resilience, executable code/data size, and runtime performance are just a few examples of features that might be in contention with each other. Reducing the need for human-intervention to resolve these contentious interactions will certainly be necessary on the path to achieving exascale computation.

ANGEL is still maturing and there are multiple avenues for future work. Firstly, a study of how different subordinate search algorithms affect the quality and efficiency of ANGEL would help us determine the problems for which it is best suited. Secondly, further investigation is necessary to determine if $N - 1$ search iterations are truly necessary. Since all objectives are being observed at all times, there exists the possibility of accelerating the convergence properties of ANGEL. Finally, there are multiple ways to parallelize the iterations of ANGEL. Finding avenues for ANGEL to execute in parallel would certainly cause it to arrive at a solution more quickly.

Acknowledgments

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program under Award Numbers ER25763 and ER26054.

7. REFERENCES

- [1] COELLO, C. A. An updated survey of GA-based multiobjective optimization techniques. *ACM Comput. Surv.* 32, 2 (June 2000), 109–143.
- [2] DEB, K., THIELE, L., LAUMANN, M., AND ZITZLER, E. Scalable multi-objective optimization test problems. In *Evolutionary Computation, 2002. CEC '02. Proceedings of the 2002 Congress on* (May 2002), vol. 1, pp. 825–830.
- [3] HAN, L., AND NEUMANN, M. Effect of dimensionality on the Nelder-Mead simplex method. *Optimization Methods and Software* 21, 1 (2006), 1–16.
- [4] JORDAN, H., THOMAN, P., DURILLO, J., PELLEGRINI, S., GSCHWANDTNER, P., FAHRINGER, T., AND MORITSCH, H. A multi-objective auto-tuning framework for parallel codes. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for* (Nov 2012), pp. 1–12.
- [5] KNOWLES, J. ParEGO: a hybrid algorithm with on-line landscape approximation for expensive multiobjective optimization problems. *Evolutionary Computation, IEEE Transactions on* 10, 1 (2006), 50–66.
- [6] MARLER, R., AND ARORA, J. Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization* 26, 6 (2004), 369–395.
- [7] MIGDALAS, A., PARDALOS, P. M., AND VÄRBRAND, P. *Multilevel optimization: algorithms and applications*, vol. 20. Springer, 1998.
- [8] NELDER, J. A., AND MEAD, R. A simplex method for function minimization. *The Computer Journal* 7, 4 (1965), 308–313.
- [9] OKABE, T., JIN, Y., OLHOFFER, M., AND SENDHOFF, B. On test functions for evolutionary multi-objective optimization. In *Parallel Problem Solving from Nature - PPSN VIII*, X. Yao, E. Burke, J. Lozano, J. Smith, J. Merelo-Guervós, J. Bullinaria, J. Rowe, P. Tiño, A. Kabán, and H.-P. Schwefel, Eds., vol. 3242 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2004, pp. 792–802.
- [10] PARKINSON, J., AND HUTCHINSON, D. An investigation into the efficiency of variants on the simplex method. *Numerical Methods for Nonlinear Optimization* (1972), 115–135.
- [11] ROIJERS, D. M., VAMPLEW, P., WHITESON, S., AND DAZELEY, R. A survey of multi-objective sequential decision-making. *Journal of Artificial Intelligence Research* 48 (2013), 67–113.
- [12] TIWARI, A., TABATABAEE, V., AND HOLLINGSWORTH, J. K. Tuning parallel applications in parallel. *Parallel Comput.* 35, 8-9 (Aug. 2009), 475–492.
- [13] VAN VELDTHUIZEN, D. A., AND LAMONT, G. B. Multiobjective evolutionary algorithm test suites. In *Proceedings of the 1999 ACM Symposium on Applied Computing* (New York, NY, USA, 1999), SAC '99, ACM, pp. 351–357.
- [14] WALTZ, F. M. An engineering approach: Hierarchical optimization criteria. *Automatic Control, IEEE Transactions on* 12, 2 (April 1967), 179–180.
- [15] ZULUAGA, M., SERGENT, G., KRAUSE, A., AND PÜSCHEL, M. Active learning for multi-objective optimization. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)* (2013), S. Dasgupta and D. McAllester, Eds., vol. 28, JMLR Workshop and Conference Proceedings, pp. 462–470.