

# Experimental Study of Global and Local Search Algorithms in Empirical Performance Tuning

Prasanna Balaprakash, Stefan M. Wild, and Paul D. Hovland

Mathematics and Computer Science Division  
Argonne National Laboratory, Argonne, IL 60439  
{pbalapra,wild,hovland}@mcs.anl.gov

**Abstract.** The increasing complexity, heterogeneity, and rapid evolution of modern computer architectures present obstacles for achieving high performance of scientific codes on different machines. Empirical performance tuning is a viable approach to obtain high-performing code variants based on their measured performance on the target machine. In previous work, we formulated the search for the best code variant as a numerical optimization problem. Two classes of algorithms are available to tackle this problem: global and local algorithms. We present an experimental study of some global and local search algorithms on a number of problems from the recently introduced SPAPT test suite. We show that local search algorithms are particularly attractive, where finding high-performing code variants in a short computation time is crucial.

## 1 Introduction

The rapid rate of innovations in computing architectures has widened the gap between the theoretical peak and the achievable performance of scientific codes [1]. Often, scientific application programmers address this issue by manually rewriting the code for the target machine, but this approach is neither scalable nor portable. Empirical performance tuning or automatic performance tuning (in short, *autotuning*) is a promising approach to address the limitations of manual tuning. This approach consists of identifying relevant code optimization techniques (such as loop unrolling, register tiling, and loop vectorization), assigning a range of parameter values using hardware expertise and application-specific knowledge, and then either enumerating or searching this parameter space to find the high performing parameter configurations for the given machine. Using this approach, several researchers have achieved considerable success in tuning scientific kernels for both serial and multicore processors [1].

In large-scale empirical performance tuning, the computation time needed to enumerate all parameter configurations in a large decision space is prohibitively expensive. Hence, effective global and/or local search algorithms that examine a tiny subset of the possible configurations are required. Typically, global algorithms can be characterized by their dynamic balance between exploration of

the search space and exploitation of the accumulated search history. They are theoretically guaranteed to find the globally best configuration at the expense of a long search time. In practice, however, they are run until user-defined stopping criteria are met. Examples include branch and bound, simulated annealing, genetic algorithms, and particle swarm optimization. In contrast, local search algorithms do not emphasize exploration and instead repeatedly try to move from a current configuration to a nearby improving configuration. Typically, the neighborhood of a given configuration is problem-specific and defined by the user or algorithm. These algorithms terminate when a current configuration does not have any improving neighbor and hence is locally optimal. Examples include the Nelder-Mead simplex, orthogonal search, variable neighborhood search, and trust region methods. The disadvantage of local search algorithms is that, depending on the search space and initial configuration, they can terminate with a locally optimal configuration that performs much worse than a globally optimal configuration.

Search problems in empirical performance tuning are defined by a specific combination of a kernel, an input size, a set of tunable decision parameters, a set of feasible parameter values, and a default/initial configuration of these parameters for use by search algorithms [2]. Several global and local search algorithms have been deployed for empirical performance tuning. Seymour et al. [9] performed an experimental comparison of several global (random search, a genetic algorithm, simulated annealing, particle swarm) and local (Nelder-Mead and orthogonal search) optimization algorithms. Similarly, Kisuki et al. [6] compared random search, a genetic algorithm, and simulated annealing with pyramid search and window search. In both these studies, the experimental results showed that the random search was more effective than the other algorithms tested. A reason is that in the tuning tasks considered, the number of high-performing parameter configurations is large and hence it is easy to find one of them. Moreover, we suspect that the adopted local search algorithms were less effective because they were not customized. Although Norris et al. [7] implemented the Nelder-Mead simplex method, simulated annealing, and a genetic algorithm in the empirical performance tuning framework *Orio*, the authors did not conduct an experimental comparison. A number of works deploy local search algorithms for empirical performance tuning. Examples include orthogonal search in *ATLAS* [11], pattern search in loop optimization [8], and a modified Nelder-Mead simplex algorithm in *Active Harmony* [10]. However, a comparison with global search algorithms was not available. From the literature, it is not clear whether local search or global search is best suited for empirical performance tuning and, in particular, under what conditions one class may be better than another.

In this paper, we focus on a setting where the available computation time for tuning is highly limited. Our hypothesis is that appropriately modified local search algorithms can find high-performing code variants in short computation times. This is based on the rationale that the exploration component of global search algorithms is less beneficial in empirical performance-tuning problems where finding high-performing configurations in short computation time is more

important than finding the optimal configuration. We conduct an experimental study of some global and local search algorithms on a number of problems from the SPAPT test suite [3]. The main contribution of the paper is empirical evidence for the effectiveness of the local search algorithms under short computation times.

## 2 Search algorithms

For global search algorithms, we consider random search, a genetic algorithm, and simulated annealing. For local search algorithms, we use the Nelder-Mead simplex method and a surrogate-based search.

**Random search** has been shown to be effective on a number of performance-tuning tasks. The parameter configurations are sampled uniformly at random from the feasible domain  $\mathcal{D}$  without replacement. At iteration  $k$ , each  $x \in \mathcal{D}$  not already sampled has probability  $\frac{1}{|\mathcal{D}|-k+1}$  of being selected as the point  $x^{(k)}$ . In the absence of other criteria, the algorithm terminates after  $|\mathcal{D}|$  iterations with the global minimum.

**Genetic algorithms** are among the most widely used global search algorithms. These algorithms follow a common framework that consists of iteratively modifying a population of configurations by applying a set of evolutionary operations such as reproduction, recombination, and mutation. Several variants exist; the best one depends on the problem at hand and the parameters of the algorithm. We use a genetic algorithm based on [4].

**Simulated annealing** is inspired by the physical process of annealing. The key algorithmic component is an annealing schedule that slowly reduces the value of a temperature parameter  $T$  so that the probability of accepting a worse configuration decreases as the search progresses [5]. The mechanism of accepting worse configurations during the search helps the algorithm escape from bad local configurations encountered in the early stages of the search.

**The Nelder-Mead simplex method** was originally developed to solve unconstrained continuous optimization problems. It works with a simplex of  $n + 1$  vertices, where  $n$  is the number of parameters. At each iteration, the simplex moves away from less promising regions of the search space using reflection, expansion, contraction, or shrink operators. We use a Nelder-Mead simplex algorithm that is customized for empirical performance tuning task; see [2] for implementation details.

**Surrogate-based search** is an algorithmic framework that uses inexpensive surrogates to approximate the computationally expensive objective. For our experiments, we consider a basic trust-region algorithm [12] that operates on discrete values. It starts by constructing a quadratic surrogate function by evaluating a few configurations. At each iteration, a configuration that minimizes the surrogate is evaluated, and the ratio between the true function value and the predicted surrogate value is used to monitor the quality of the surrogate. When the surrogate is accurate enough, the trust region is expanded; otherwise, the region is contracted, and a promising neighbor of the current configuration is evaluated to improve the surrogate.

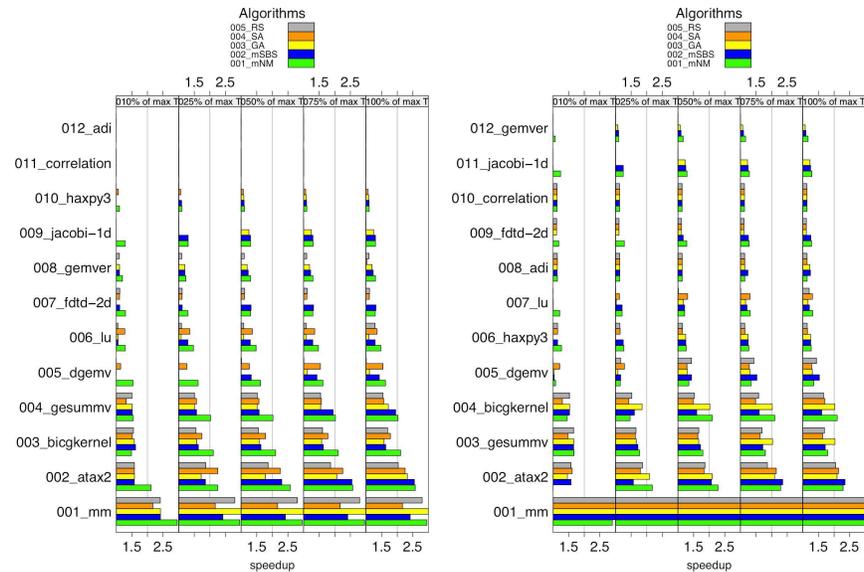
### 3 Numerical experiments

We evaluate the algorithms on problems from the SPAPT test suite [3], a collection of extensible and portable search problems in automatic performance tuning. These problems are implemented in an annotation-based language that can be readily processed by Orio [7]. Originally, the SPAPT problems had integer and binary parameters (scalar replacement, array copy, loop vectorization, and OpenMP) with both bound and algebraic constraints. Since the focus of our study is on bound-constrained problems with integer parameters only, we removed all algebraic constraints and binary parameters from the problems. The numerical parameters include loop unroll/jamming  $\in [1, \dots, 50]$ , cache tiling  $\in [1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048]$  (treated as  $[1, \dots, 12]$ ), and register tiling  $\in [1, \dots, 32]$ . The number of parameters  $n_i$  ranges between 8 and 38, and the size of search space  $|\mathcal{D}|$  ranges between  $5.31 \times 10^{10}$  and  $1.24 \times 10^{53}$ . Of the 18 problems in the SPAPT test suite, we use only 12. On the remaining 6 problems, since the algebraic constraints are required for the correctness of the transformation, we did not use it.

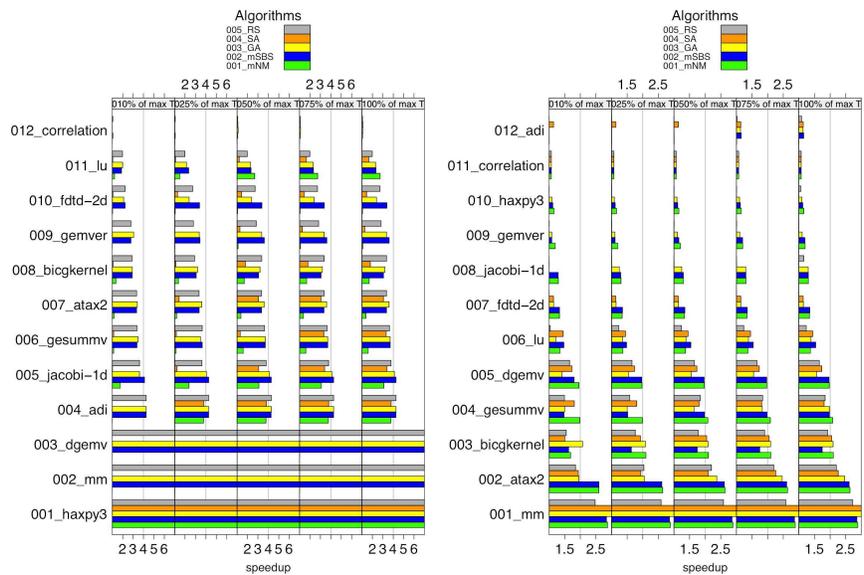
Random search (RS), the genetic algorithm (GA), simulated annealing (SA), modified Nelder-Mead simplex (mNM), and modified surrogate-based search (mSBS) were implemented and run in MATLAB version 7.9.0.529 (R2009b). We adopted the default parameter values for all the algorithms. Experiments were carried out on dedicated nodes of Fusion, a 320-node cluster at Argonne National Laboratory, comprising 2.6 GHz Intel Xeon processors with 36 GB of RAM, under the stock Linux kernel version 2.6.18 provided by RedHat.

We considered the objective value  $f(x)$  at a parameter configuration  $x$  as the average computation time over 10 generated code runs. Other objective functions can be adopted, such as the median or minimum; see [3] for a discussion. For the initial configuration from which the algorithms start, we set each parameter to its lower bound. This corresponds to a code variant without any transformation. We used 100 code evaluations as the stopping criterion for each algorithm. Given a parameter configuration, a code evaluation consists of code transformation, compilation, and execution. For the size of the search space that we have, this corresponds to the evaluation of only  $8.05 \times 10^{-50}\%$  ( $|\mathcal{D}|=1.24 \times 10^{53}$ ) to  $0.00000018\%$  ( $|\mathcal{D}|=5.31 \times 10^{10}$ ) of the total configurations.

Figure 1(a) shows a bar chart of the speedups at different time intervals. We compute “ $x\%$  of max T” is computed as follows: For each problem, max T is the maximum elapsed time that any of the five algorithms took to complete 100 evaluations. The speedups obtained by each algorithm after 10%, 25%, 50%, and 100% of the max T is computed and shown in the figure. From the speedups obtained at these intervals, we observe that the two local search algorithms, mNM and mSBS, obtain high-quality configurations in short computation time. The main advantage here comes from the time required for the algorithms to complete 100 code evaluations. RS and GA require longer search times because they spend more time exploring the domain and tend to be slower than mNM and mSBS. The performance advantage of mNM and mSBS comes from the fact that the time per evaluation tends to be shorter once a good configuration has been



(a) Default initial configuration; default input size; 100 function evaluations (b) Default initial configuration; large input size; 100 function evaluations



(c) Poor initial configuration; default input size; 500 function evaluations (d) Default initial configuration; default input size; 500 function evaluations

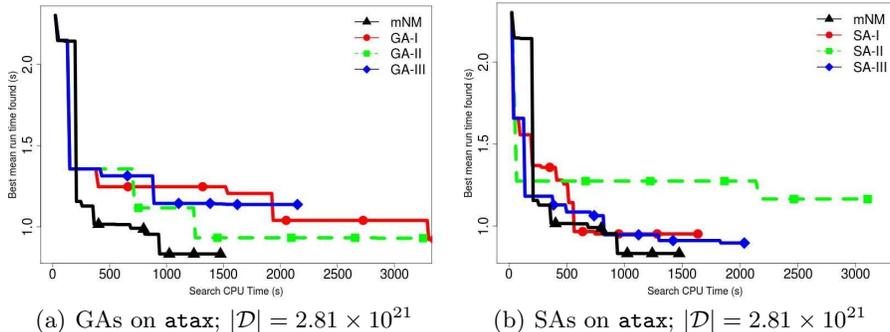
**Fig. 1.** Speedups obtained by each algorithm as a function of % of the budget.

found. On 9 of 12 problems, we found that the local algorithms outperformed the global algorithms. The observed speedups are between 1.15 and 3.0, respectively. On `adi` and `correlation`, we cannot detect a significant speedup.

Under the same computation budget of 100 code evaluations, we tested the behavior of the algorithms on larger input sizes (the size of the arrays and matrices in the kernels) by doubling the input size for each problem. The results are shown in Figure 1(b). Although the times to complete 100 code evaluations are larger than those observed with smaller input sizes, the trend in the behavior of the algorithms is similar: the local search algorithms obtain high-performing code variants in short computation time. Out of 12 problems, on 8 problems the local search algorithms are better than the global search algorithms. On `correlation`, we cannot detect a significant difference between the results of the global and local search algorithms. Although mNM and mSBS find high-quality configurations in short computation time, on `gessumv`, given enough time GA obtains a better configuration than mNM and mSBS. On `mm` and `correlation`, we cannot detect a significant difference between the results of the global and local search algorithms.

Figure 1(c) shows the results when the starting point is set to the upper-bound values. From the exploratory studies, we found that the initial configurations with lower-bound values are reasonably good starting points and that those at the upper bounds are extremely poor. We found that mNM and SA tend to be sensitive to the starting point and obtain poor results. These algorithms also required longer search times because the parameter configurations closer to the upper bounds have longer transformation time and consequently longer compile time. Whereas SA tries to escape from the nonpromising region, mNM stagnates, spending most of the search time exploring the neighborhood of the current configuration. We found mSBS to be less sensitive than mNM or SA to the starting point because it uses randomly sampled configurations within a larger initial neighborhood to form the initial surrogate. GA uses the initial configuration only as an individual of the population in the first iteration. Since RS is independent of the starting point, it found better code variants than did mNM and SA in short computation times. The results show that the poor starting points significantly reduce the effectiveness of the local search algorithms. Out of 12, only on 6 problems did the local search algorithms, in particular, mSBS, outperform the global search algorithms. We also used the center of the hyperrectangle  $\mathcal{D}$  as a starting point. The results observed are similar to those with lower bounds as in Figure 1(c), local algorithms being better than the global algorithms despite a slightly worse starting value than the lower bounds.

Figure 1(d) illustrates the behavior of the algorithms using a slightly larger computation budget (500 code evaluations) as the stopping criterion. The algorithms start from initial configurations in which each parameter is set to its lower-bound value. Global search algorithms benefit from a larger number of iterations. On 7 out of 12 problems local search algorithms dominate global search algorithms, but the difference in the speedups between global and local algorithms is smaller than that observed with 100 evaluations. Although local



**Fig. 2.** Best objective value obtained by each algorithm as a function of search time. Each algorithm is allowed to perform 100 function evaluations. Markers are placed at every 20 evaluations.

search algorithms find high-quality code variants in short times, they spend the search effort in exploring the neighborhood of a local configuration to certify local optimality.

To further test that the exploration component is the major factor affecting the performance of global search algorithms, we reduced their degree of exploration. Specifically, for GA and SA, we reduced the values of the mutation parameter  $\mu$  and starting temperature parameter  $T$ , respectively. We used three GAs: GA-I (default  $\mu = 0.5$ ), GA-II ( $\mu = 0.1$ ), and GA-III ( $\mu = 0.001$ ). Similarly for SA, we used SA-I (default  $T = 1.0$ ), SA-II ( $T = 0.1$ ), and SA-III ( $T = 0.001$ ). Figures 2(a) and 2(b) illustrate the results of the algorithms on `atax` for 100 code evaluations. The default lower-bound configuration is used as a starting point. The results of our study show that reducing the exploration in global search algorithms is beneficial but the appropriate reduction depends on the algorithm characteristics, the problem, and the starting point. GA-I and GA-II obtain configurations with similar runtime, but the latter obtains this configuration in a shorter period of time (1200 CPU-seconds). However, an extremely small degree of exploration in GA-III leads to stagnation. In contrast, although slightly slower, SA-III obtains a better configuration than do SA-I and SA-II. Our conjecture is that given a good starting point, SA with a very low degree of exploration can be effective.

## 4 Conclusion

We investigated the issue of global versus local search in empirical performance tuning under short computation times. We tested illustrative global and local algorithms on bound-constrained search problems with integer parameters. We used different initial configurations, input sizes, and stopping criteria. The results show that (1) the exploration capabilities of global search algorithms are less useful; (2) given good initial configurations, local search algorithms can find high-performing code variants in short computation time; and (3) poor initial configurations can significantly reduce the effectiveness of both global and local search algorithms that are sensitive to the starting point. From the

results, we conclude that when the available tuning time is severely limited, carefully customized local search algorithms are promising candidates for empirical performance-tuning problems that have integer parameters and bound constraints.

Our future work includes the following: (1) problem-specific techniques to handle binary parameters and constraints for both global and local search algorithms, (2) effective restart and multi start strategies for local search to escape from poor local configurations, (3) global algorithms that automatically adopt exploration and exploitation parameters, (4) tuning of parallel scientific codes using search algorithms, and (5) analysis of the impact of different target machines on various performance objectives.

### Acknowledgments

This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357. We are grateful to the Laboratory Computing Resource Center at Argonne National Lab.

### References

1. D. Bailey, R. Lucas, and S. Williams, editors. *Performance Tuning of Scientific Applications*. Chapman & Hall, 2010.
2. P. Balaprakash, S. M. Wild, and P. D. Hovland. Can search algorithms save large-scale automatic performance tuning? In *Int. Conf. on Computational Science*, 2011.
3. P. Balaprakash, S. M. Wild, and B. Norris. SPAPT: Search problems in automatic performance tuning. Preprint ANL/MCS-P1872-0411, Argonne National Lab, 2011.
4. A. Chipperfield and P. Fleming. The MATLAB genetic algorithm toolbox. In *IEE Colloquium on Applied Control Techniques Using MATLAB*, 1995.
5. S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.
6. T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O’Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proc. Int. Conf. on Parallel Arch. and Compilation Techniques*, Washington, DC, 2000.
7. B. Norris, A. Hartono, and W. Gropp. *Annotations for Productivity and Performance Portability*, pages 443–461. Computational Science. CRC Press, 2007.
8. A. Qasem, K. Kennedy, and J. Mellor-Crummey. Automatic tuning of whole applications using direct search and a performance-based transformation system. *Supercomputing*, 36(2):183–196, 2006.
9. K. Seymour, H. You, and J. Dongarra. A comparison of search heuristics for empirical code optimization. In *Proc. 2008 IEEE Int. Conf. on Cluster Computing*, pages 421–429, 2008.
10. A. Tiwari, C. Chen, C. Jacqueline, M. Hall, and J. K. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *Proc. of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12, Washington, DC, 2009.
11. R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proc. 1998 ACM/IEEE Conf. on Supercomputing, SC ’98*, pages 1–27, Washington, DC, 1998.
12. S. M. Wild. MNH: A derivative-free optimization algorithm using minimal norm Hessians. In *Tenth Copper Mountain Conference on Iterative Methods*, 2008.