

# Application-Level Automatic Performance Tuning on the Single-Chip Cloud Computer

Victor Pankratius

Karlsruhe Institute of Technology

76131 Karlsruhe, Germany

pankratius@kit.edu, www.victorpankratius.com

Sven Blaese

Karlsruhe Institute of Technology

76131 Karlsruhe, Germany

sven.blaese@student.kit.edu

**Abstract**—Improving application performance is the main motivation to switch to manycore hardware and parallelize all kinds of software. Intel’s Single-chip Cloud Computer (SCC) offers a testbed with 48 general-purpose cores on one chip as a first step towards chips with even more cores. However, a current difficulty is that programmers are responsible for controlling a variety of performance-affecting parameters that are interdependent and that are influenced by both hardware and software. Thus, tuning parallel applications on the SCC is far from trivial, and manual approaches are tedious. To address these problems, this paper is the first to introduce an application-level automatic performance tuning approach on the SCC. Programmers identify performance-impacting parameters within their software applications and let an external auto-tuner search for the best configuration. We discuss and evaluate several tuning algorithms, including HyDES, our own approach that combines Differential Evolution and Nelder-Mead Simplex methods in a novel way. First results on parallel compression and other applications show that HyDES can significantly boost performance. In addition, our measurements reveal that good parameter configurations can be non-intuitive; in certain scenarios, existing programs such as MPIBZIP do not even allow users to set the optimal performance parameter configuration from the command line. Our auto-tuning approach greatly simplifies the application performance optimization process. It is easy to use and has the potential to become a standard approach for large number of SCC programmers.

## I. INTRODUCTION

Multicore and manycore chips are here to stay, so programmers need to develop parallel software to exploit the hardware potential. Average programmers with little experience in parallel programming now face a spectrum of additional difficulties, such as writing programs that are correct and at the same time perform well on all available parallel platforms. However, already on one single platform, parallel application performance tuning is far from trivial. A key reason is that parallel applications typically depend on a variety of inter-related software parameters, hardware parameters, and input parameters that influence performance in ways that can be too complex to model with acceptable effort. As a consequence, many programmers resort to manual trial-and-error when it comes to tuning (e.g., to determine which degree of parallelism leads to the best compute performance, which data partitioning has optimal cache exploitation, etc.). This tedious process is typically repeated every time when a program is ported to a new platform.

Intel’s experimental Single-chip Cloud Computer (SCC) offers a glimpse into the future on how it would be like to program a manycore computer with 48 general-purpose cores. With an increasing number of cores, hardware needs to adapt and make tradeoffs, which leads to solutions that differ from today’s mainstream. This is why the SCC’s architecture offers programmers even more freedom than many other multicore architectures, which has significant consequences for programmers. More control in software is good to squeeze out the last percent of performance, but the downside is that application tuning becomes even more difficult than it is already on current shared-memory multicore platforms. For example, every SCC core can boot an own operating system image (where each image might be optimized differently) and communicate via message passing with other cores, just as in a regular distributed system. Communication performance depends, among others, on which cores communicate and how the software assigns work to cores. Inexperienced programmers might be overwhelmed by such details and fail to achieve good performance, while experts might have to invest large amounts of time in performance optimization instead of moving on with other features. Software engineers clearly need automation support for performance optimization to be more productive on the SCC.

We tackle these problems in this paper, which is the first paper to present an application-level automatic performance tuning approach on the SCC. In particular, we make the following novel contributions. We introduce an SCC auto-tuner that automatically searches for good performance configurations for every tunable SCC application. To create tunable applications, we exploit programmer’s knowledge and let programmers define application performance parameters that the auto-tuner should configure. For example, parameters may include the maximum number of processes to generate, block sizes for data structures, number of pipeline stages to create, and choices of predefined algorithm variants for a particular program task. Our technique leverages parallelism on higher abstraction levels and complements other performance optimizations (e.g. compiler optimizations). We introduce HyDES, a novel search algorithm based on Nelder-Mead Simplex and Differential Evolution methods, which has been specifically developed and tested for the SCC. We experimentally compare our approach with others using standard benchmarks and show

that our approach leads to significantly better performance. Our SCC auto-tuner is able to improve the performance even for parallel benchmark programs that are already optimized with other techniques. Case studies reveal that well-performing configurations can be non-intuitive.

The paper is organized as follows. Section II details the problem specification and our application context. Section III introduces auto-tuning strategies for parallel applications. Section IV presents experimental evaluations on the SCC. Section V contrasts related work. Section VI provides a conclusion.

## II. PROBLEM SPECIFICATION AND CONTEXT

Our optimization seeks to minimize the application run-time

$$f(\vec{x}), \text{ where } \vec{x} = (x_1, \dots, x_N)$$

is a vector of tunable application parameters. Each parameter  $x_i$  has values out of a predefined valid range  $[x_{il}, x_{ih}] \subset \mathbb{R}$ .

We assume that developers write programs in such a way that the programs are configurable, i.e.,  $\vec{x}$  is configurable from the command line or the program communicates the addresses of configurable variables to the auto-tuner. For now, we assume for simplification that program inputs (e.g., files used in computations) are additional implicit parameters; we treat program tuning with different inputs as own optimization problems. The function  $f$  depends on the specific properties of each program and is typically unknown or difficult to model.

The purpose of our auto-tuner is to empirically find good configurations of  $\vec{x}$  that minimize the function  $f$ , which in our examples corresponds to finding the lowest parallel application run-time. The complete search space consists of the cartesian product of all parameter domains. i.e.,  $dom(x_1) \times dom(x_2) \times \dots \times dom(x_N)$ . Trying out the entire search space is obviously impractical for most application scenarios. Our auto-tuner thus works iteratively to explore the search space: It generates values for parameters using the algorithms described in the next section, executes the programs, measures performance, and uses the feedback to find more promising parameter values. The process stops after a defined number of iterations or if other termination criteria are satisfied (e.g., run-time is below a certain threshold).

### A. Homogeneous and Heterogeneous Tuning

For each auto-tuning algorithm, we developed and implemented two versions that distribute the tuning process among SCC cores in different ways.

In *homogeneous tuning* (Figure 1), there is a master core that runs the tuning algorithm, computes the parameter values of  $\vec{x}$ , and assigns the same  $\vec{x}$  to all worker cores. The worker cores then process a program's tasks (which are all split up equally) in parallel. Considering parallel compression with BZIP as an example, all cores would compress blocks of a file using the same file block size. Updates to the block size are performed by the master between complete program runs.

In *heterogeneous tuning* (Figure 2), every core is its own master and computes its own  $\vec{x}$ . When a core is done, it

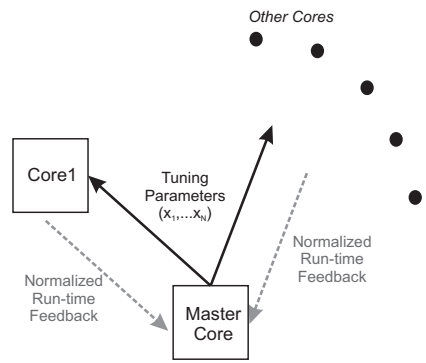


Fig. 1. Principles of homogeneous tuning on SCC cores.

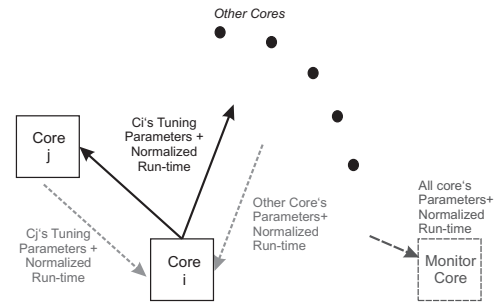


Fig. 2. Principles of heterogeneous tuning on SCC cores.

broadcasts  $\vec{x}$  and the run-time feedback (normalized w.r.t amount of work) to all other cores. When finished, the cores may adapt the new parameters if they are better than their own. A monitor core collects global statistics about the best parameter configuration so far. This approach parallelizes and exchanges information about the optimization process itself and may converge faster. However, it might not be applicable to all sort of programs, e.g., when differing parameter values are not allowed for different workers. This is why we have two versions of tuning, so we can resort to homogeneous tuning if necessary. Heterogeneous tuning for parallel compression, for example, has different cores working on different file blocks, but each core uses different file block sizes; block sizes are adapted while the application is running when new work is assigned to cores.

## III. AUTO-TUNING STRATEGIES ON THE SCC

### A. Random Tuning – A Comparison Baseline

This strategy serves as a comparison baseline for other tuning algorithms, including ones from the literature. A good tuning algorithm should be able to beat Random on the same number of iterations to justify the implementation effort.

1) *Homogeneous tuning*: In each iteration, we generate a random vector  $\vec{x}$  with elements  $x_i \sim U(x_{il}, x_{ih})$  with uniform random distribution. The same vector is used by every worker core. The master keeps the vector that leads to the best application run-time after every tuning iteration.

2) *Heterogeneous tuning*: Each worker core configures its random vector on its own and logs the best configuration.

When a core completes its work, it sends the best configuration so far to the monitor core. The monitor core keeps the best configuration found by any worker core.

### B. Nelder-Mead Simplex

The principle of this algorithm is well-known in the literature [1], [2], [7], which is why we implement it for comparison.

1) *Homogeneous tuning*: The algorithm has an elaborate scheme with several case differentiations whose details are beyond the scope of this paper, so we briefly sketch the ideas. A simplex  $s \in (\mathbb{R}^N)^{N+1}$  is the simplest polygon in  $N$  dimensions (e.g., a triangle for  $N = 2$ ). For an  $N$ -dimensional search space, the algorithm initializes  $N + 1$  vertices with random values. Then, each iteration evaluates the points of the current simplex and moves it to more promising locations. There are several rules to move simplex points, such as reflection, expansion, contraction, and reduction, which are described in [1], [2]. For these rules, our implementation uses the common values of  $\alpha = 1$ ,  $\beta = 0.5$ ,  $\gamma = 2$ .

2) *Heterogeneous tuning*: Each worker core runs the entire algorithm on its own and generates a random initial simplex. After a core finishes the optimization, it communicates the best result so far to the monitor core that gathers the best configuration from all cores.

### C. Differential Evolution

Literature shows that Nelder-Mead is not powerful enough when it comes to escaping global minima, as it might get trapped in local minima [3]. Differential Evolution is a more promising heuristic that compensates this shortcoming. It works even on non-linear and non-differentiable functions, and it is less computationally intensive than other methods that have the same goal [3]. In each iteration, Differential Evolution works on a population of individuals (i.e., parameter configurations or vectors, respectively), evaluates the best individuals, and replaces individuals in new population generations by using several operators as described next.

1) *Homogeneous tuning*: Our approach is based on [3] and starts with a population of  $n_{pop}$  configurations, where each individual has values randomly chosen from each parameter range. The population size  $n_{pop}$  remains constant during the entire optimization process. A mutation operator creates new configurations by taking the difference of two random population vectors, weighing it by a factor  $F = \sqrt{\frac{1}{n_{pop}} - \frac{\alpha}{2n_{pop}}}$  [4], and adding the result to a third randomly chosen population vector. To increase diversity, the mutated vector is mixed with yet another randomly chosen population vector  $v_{target}$  by taking an element from the mutated vector with probability  $\alpha = 0.7$ , and from  $v_{target}$  with  $1 - \alpha$ , resulting in the vector  $v_{trial}$ . The program to optimize is run with the configuration of  $v_{target}$  and  $v_{trial}$  on all cores, and the better-performing configuration is kept in the new population, which is used as a starting point in future iterations.

2) *Heterogeneous tuning*: One or more individuals are represented by one core. Each core works with a different configuration of individuals. After each individual has been evaluated, each core selects the best individual so far and propagates the configuration to all other cores (using `allgather` from `RCCE_comm`). Then, each core generates a new individual as described for homogeneous tuning, and the process restarts.

### D. HyDES: Hybrid Differential Evolution Simplex

Our new approach combines the best of Nelder-Mead and Differential Evolution methods. As shown later, it is effective for application-level auto-tuning on the SCC.

1) *Homogeneous tuning*: As in Differential Evolution, we start with a population of  $n_{pop}$  vectors (i.e., configurations). One randomly chosen vector  $\vec{p} = (p_1, \dots, p_N)$  out of this population is used to generate a simplex. We employ the following schema to generate  $N$  additional simplex points and ensure that the simplex stretches appropriately in the search space: For every dimension  $i$ , we compute  $d_i = (max_i - min_i)/4$ . If  $max_i - p_i \leq p_i - min_i$  then we multiply  $d_i$  by  $-1$  to achieve a larger stretching of the simplex. Then, we generate new vectors by adding  $d_1$  to  $p_1$ ,  $d_2$  to  $p_2$ , and so on. So from  $p$ , each new simplex vector has a distance of  $d_i$  in dimension  $i$ . In case that a simplex point is above any  $max_i$  or below any  $min_i$  of a dimension  $i$ , we correct the value in this dimension by moving the vector by a random value into the valid search space. This random displacement is computed using a normal distribution that has its mean at the violated boundary and the standard deviation  $(max_i - min_i)/20$ . With the resulting simplex, we perform optimization as described by Nelder-Mead, which stops when the maximum distance between any simplex vectors is less than a predefined  $\epsilon$ . The result replaces the initially selected individual  $p$ . Then, Differential Evolution continues with the new population in which the new result is used to generate other vectors as in Section III-C. Nelder-Mead is applied on future populations with 5% probability, otherwise Differential Evolution continues as described.

2) *Heterogeneous tuning*: Each core is responsible for one individual and initialized by Differential Evolution. Each core communicates its individual to all other cores. As an extension to Figure 2, the monitor core also acts as a master to dictate to all other cores when to apply Nelder-Mead and when to apply Differential Evolution. When Nelder-Mead is applied, the master randomly chooses one core to monitor. When that core finishes its Nelder-Mead optimization, the optimization process in all other cores is stopped as well. All cores replace their old individual by the new individual and broadcast their new result to all cores.

## IV. EXPERIMENTAL EVALUATION ON THE SCC

### A. Setup

We employ the latest available Intel SCC with 48 cores (16KB L1, 256KB L2 cache per core, 16KB message passing buffer per tile). Cores are clocked at 533 MHz, routers at 800 MHz, and memory at 800 MHz. Each core runs an own

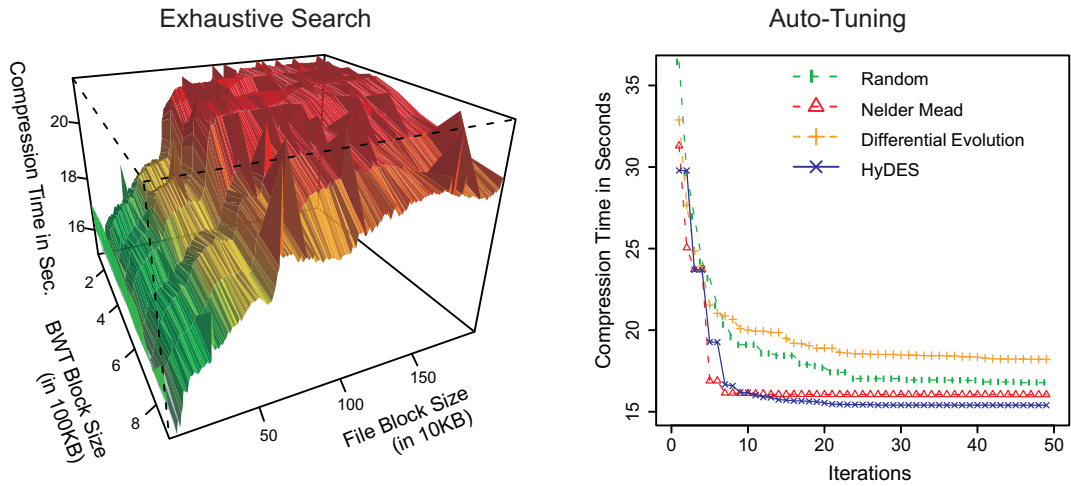
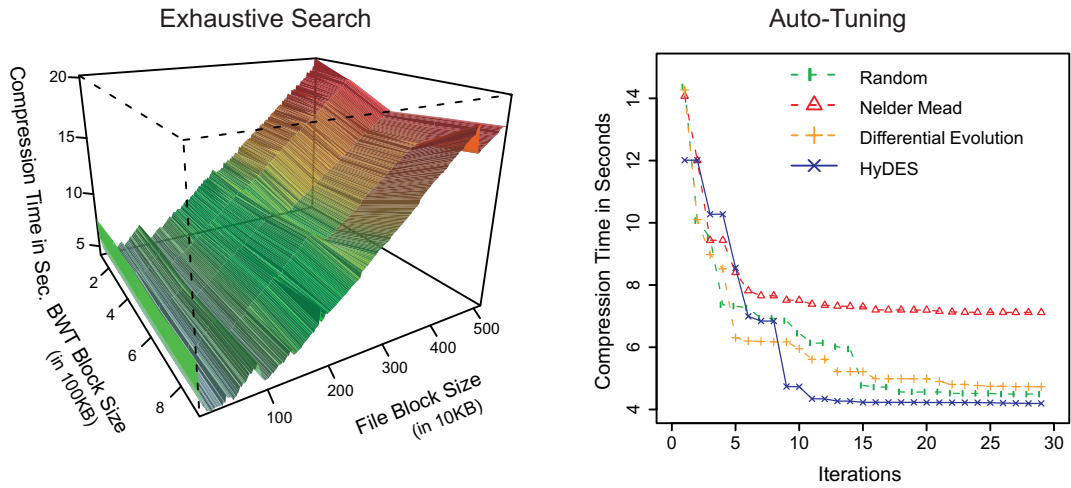


Fig. 3. Comparison of parallel compression tuning results with homogeneous tuning. Lower values are better.

image of Linux kernel 2.6.16. The compilers used are `icc 8.1.038` for XHPL benchmarks (with `MKL 8.1.1.004`) and `gcc 3.4.5` for MPIBZIP (with `RKMPI` and `RCCE`). Due to space limitations, however, we can present just an excerpt of all evaluation results.

### B. Auto-Tuning SCC Applications

To test our approach on a real application, we made the MPIBZIP compression program configurable for two important parameters (Burrows-Wheeler-Transform (BWT) block size and file block size). Figure 3 shows results with homogeneous tuning which illustrate that our HyDES approach finds the best performance configurations already after 10 program executions. The exhaustive search in the 3D graphs shows that the best performance configuration is non-intuitive; for example, with 6 cores (1 master and 5 slave cores), distributing a 5 MB file intuitively as 1 MB chunks per core is worse than

the optimum of 170 KB-sized chunks per core (also found by the auto-tuner). When using a 46 MB input file and all 48 cores, the optimum chunk size per core is 50 KB. Interestingly, MPIBZIP does not even allow users to set such a low value from the command line, so auto-tuning not only paid off, but also pointed to software improvements.

In practice, parameter tuning can be done for clusters of inputs (e.g., ranges of file sizes) when the program is customized to a platform; this yields good parameters that are representative for a class of inputs, so tuning won't have to be repeated for every single input.

### C. Stress-Test Evaluations

In addition to real applications, we used common stress-test functions from the optimization literature [5], [6] to evaluate the effectiveness of the aforementioned tuning algorithms in difficult scenarios.

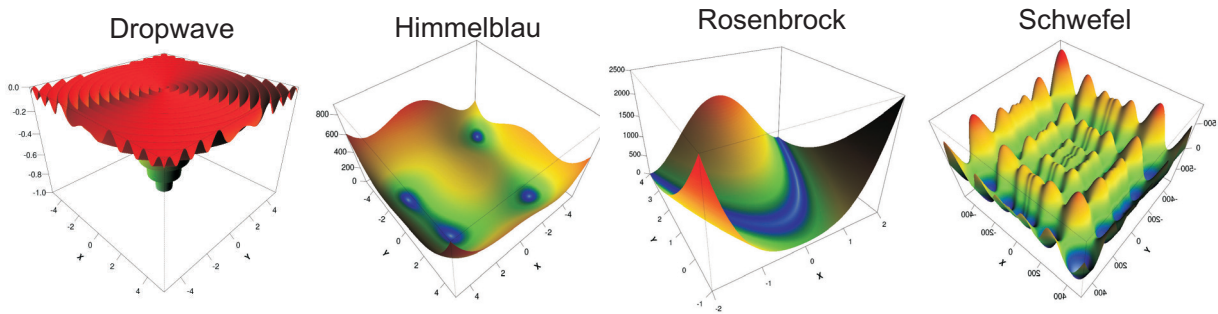


Fig. 4. Common benchmark functions [5], [6] used for optimization stress-testing.

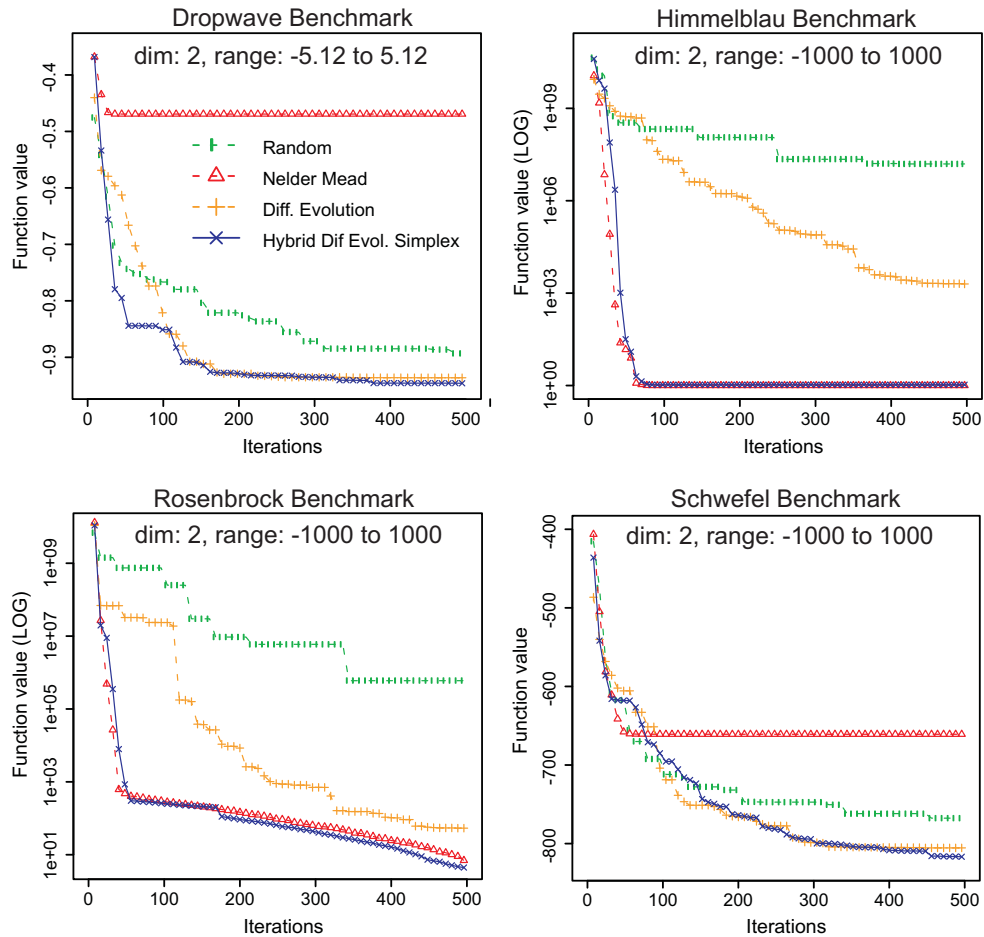


Fig. 5. Tuning results with homogeneous tuning on common optimization stress-test benchmark functions (see Fig. 4). Lower values are better.

Figure 4 exemplifies the shapes of these multidimensional functions for three dimensions. We implemented parameterized SCC test programs whose run-time performance behaves according to these functions.

Figures 5 and 6 show experimental evaluations of the function benchmarks with homogeneous and heterogeneous tuning. It is remarkable that our HyDES approach beats all other approaches in all scenarios in finding better performance configurations, while at the same time the speed of convergence (measured by number of iterations) is good as well.

The homogeneous scenarios in Figure 5 use 20 individuals for evolution. The heterogeneous scenarios in Figure 6 employ a pre-specified number of cores (10, 15, and 48 cores). With heterogeneous tuning, convergence is faster (i.e., fewer iterations are needed) even on a larger problem (4D Rosenbrock). This can be explained by the fact that the heterogeneous tuning approach parallelized the tuning process, as explained earlier.

Note that HyDES combines the best of both worlds: It avoids problems that single Nelder-Mead or Differential Evolution would run into. Sometimes Differential Evolution is

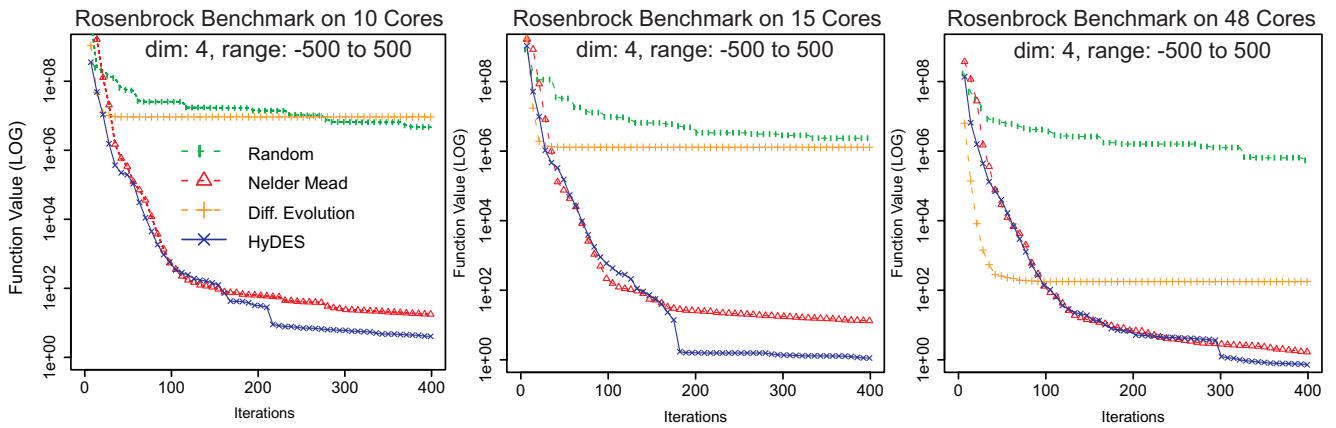


Fig. 6. Rosenbrock benchmark with heterogeneous tuning. Lower values are better.

better than Nelder-Mead or vice-versa, but HyDES always ends up better than any of the latter.

#### D. Auto-Tuning a Common Benchmark

Finally, we also tuned the XHPL (Linpack) benchmark in 11 dimensions (i.e., with 11 tuning parameters). Due its application nature, we employed homogeneous tuning. The results are shown in Figure 7.

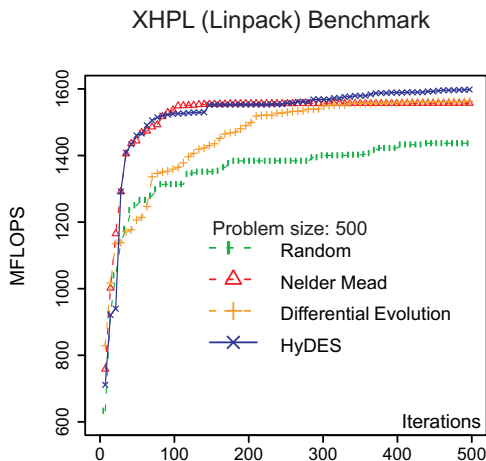


Fig. 7. Auto-Tuning the XHPL (Linpack) benchmark with homogeneous tuning (higher values are better).

The chart demonstrates the superior MFLOPS achievement of our HyDES approach. It also visualizes that systematic tuning is significantly better than using random performance configurations.

#### V. RELATED WORK

Related work on auto-tuning on the SCC is scarce, but this is because the SCC has appeared just recently and still is an experimental system. It is conceivable to adapt other auto-tuners that work on clusters (e.g., [7]) to the SCC. However, we already include a comparison with [7], as that particular tuner is based on Nelder-Mead optimization. Other tuners that use new optimization approaches in the future can

be compared via our random tuning baseline. Most of the literature on optimization [1], [2], [3], [4], [5] focuses on specific classes of algorithms that are inappropriate to use in our SCC context. For example, many algorithms either don't fit to our problem or assume that we can take arbitrarily many samples (potentially infinitely many), which translates to running a program infinitely often. Hybrid algorithms, such as our HyDES, received little attention so far.

#### VI. CONCLUSION

Automatic performance tuning makes performance engineering on the SCC less tedious for software developers. As a new aspect, our approach targets general SCC applications, not just scientific numerical applications. The results show for compression and various stress-test applications that our novel tuning approach systematically leads to better parallel performance. Auto-tuning found non-intuitive performance configurations and revealed that existing applications, such as MPIBZIP, didn't even allow users to manually set the best-performing configurations from the command line. Our technique has the potential to become standard for the SCC, as it makes application development and tuning easier.

*Remarks and Acknowledgments.* The first author is the principal investigator in the "Software Engineering for SCC Parallel Programs" project in collaboration with Intel. We thank Intel for providing us with access to the SCC.

#### REFERENCES

- [1] R. R. Barton and J. S. Ivey, Jr., "Modifications of the nelder-mead simplex method for stochastic simulation response optimization," in *Proc. IEEE WSC '91*, 1991.
- [2] J. A. Nelder and R. Mead, "A simplex method for function minimization" *Computer Journal*, vol. 7, 1965.
- [3] R. Storn and K. Price, "Differential evolution - a simple and efficient heuristic for global optimization over continuous spaces" *J. of Global Optimization*, vol. 11, 1997.
- [4] D. Zaharie, "Critical values for control parameters of differential evolution algorithm" in *Proc. 8th MENDEL intl. conf. on soft computing*, 2002.
- [5] F. Neri and V. Tirronen, "Recent advances in differential evolution: a survey and experimental analysis" *Artificial Int. Rev.*, vol. 33, 2010.
- [6] L. Schoeman and A. P. Engelbrecht, "Containing particles inside niches when optimizing multimodal functions" in *Proc. SAICSIT '05*, 2005.
- [7] C. Tapus et al. "Active Harmony: Towards Automated Performance Tuning", in *Proc. SC'02*, 2002