SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis | 979-8-3503-5554-3/24/$31.00 ©2024 IEEE | DOI: 10.1109/SCW63240.2024.00131

Evaluating Tuning Opportunities of the LLVM/OpenMP Runtime

Smeet Chheda*, Gaurav Verma*, Shilei Tian*, Barbara Chapman*, Johannes Doerfert†
*Stony Brook University, USA, schheda@cs.stonybrook.edu, {gaurav.verma, shilei.tian, barbara.chapman}@stonybrook.edu
†Lawrence Livermore National Laboratory, USA, doerfert1@llnl.gov

***Abstract*—Tuning parallel applications on multi-core architectures is an arduous task. Several studies have utilized autotuning for OpenMP applications via standardized user-facing features, namely number of threads, thread placement, binding and scheduling policy. However, they fall short on utilizing the additional parameters provided by an OpenMP implementation.**

In this paper, we analyze OpenMP application runtime through an exhaustive exploration of all relevant configuration options of the LLVM/OpenMP runtime.

Our findings allow to identify trends in tuning potential, architecture-aware tuning suggestions, and good default configurations per architecture. We will open-source the 240,000 unique samples collected during experiments for use by the community. These runs have been conducted on three different CPU architectures vital in the HPC and datacenter community. Choice of applications includes popular benchmark suites and microbenchmarks namely, NAS Parallel Benchmarks, Barcelona OpenMP Task Suite, XSBench, RSBench, SU3Bench and LULESH.

We employ the Linear Models class of Machine Learning algorithms to perform analysis, explain, and form qualitative relations between features comprising of the underlying architecture, application, input size, number of threads, and considered environment variables. This is further used to recommend different configurations given an application type/architecture.

***Index Terms*—HPC, parallel programming, tuning, machine learning**

I. Introduction

The OpenMP API has evolved from supporting simple fork-join shared-memory parallelization to additional forms of parallelism not strictly limited to shared memory. This evolution has occurred over nearly three decades encompassing extensions and additions to the specification. OpenMP now includes support for not only software-defined parallelism types such as structured (loop-based) and unstructured (asynchronous task-based parallelism), but also hardware-implemented parallelism such as SIMD (single instruction multiple data) and accelerators [1]. These extensions are available to an application developer by a rich set of directives in the C, C++, and Fortran base languages. Due to its ease of use, program and performance portability across heterogeneous architectures, most compilers (open source and proprietary) provide support for OpenMP. While the standardized API exposes various tuning opportunities to users, the actual implementations are often even more configurable. In this paper, we look at the entire set of environment variables controlling the LLVM/OpenMP CPU runtime that influence the execution of parallel applications.

OpenMP applications can be configured via pragmas, library routines, and environment variables. For standardized features, these methods influence the value of Internal Control Variables (ICVs) which control different aspects of the OpenMP runtime. For non-standardized, implementation-defined features, runtimes generally use environment variables as a control mechanism. We, therefore, focus on environment variables to influence the LLVM/OpenMP runtime behavior.

The problem our study addresses is the identification of important environment variables and the selection of their values to improve performance of parallel applications. In the LLVM/OpenMP runtime various aspects can be influenced, including ICV values, like the maximal number of threads operating in parallel, thread placement, binding, and scheduling policy, but also the alignment of internal data structures and the reduction algorithm. It is easy to set an environment variable during execution but deciding the value for all possible variables is not trivial. Running applications for all combinations is often prohibitively expensive. This means tuning for real-world applications needs to be guided to avoid the full search space exploration which grows combinatorially in the number of choices. Further, as we will show, the various configuration options are not equally important and time is best spent trying only selected values for a subset of the environment variables.

To guide users we studied the effect of performance-critical environment variables that influence the LLVM/OpenMP runtime by collecting over 240,000 unique samples on three CPU architectures. We then analyze the results statistically to identify trends and determine the most important features to drastically reduce the search space while preserving the optimization potential. Finally, we provide recommendations based on application type/architecture that have the potential of outperforming the default configuration.

Prior studies have primarily looked at tuning the number of threads, thread placement, binding, or scheduling policies. These are important factors to consider for improving performance, as evident by their standardized ICVs. However, we take a step further by including all implementation defined environment variables that may affect the performance of an application. While implementation-specific environment variables are not generally portable, the popularity of LLVM/OpenMP library and adoption by other compilers makes this study useful for almost all HPC systems.

979-8-3503-5554-3/24/$31.00 ©2024 IEEE
DOI 10.1109/SCW63240.2024.00131

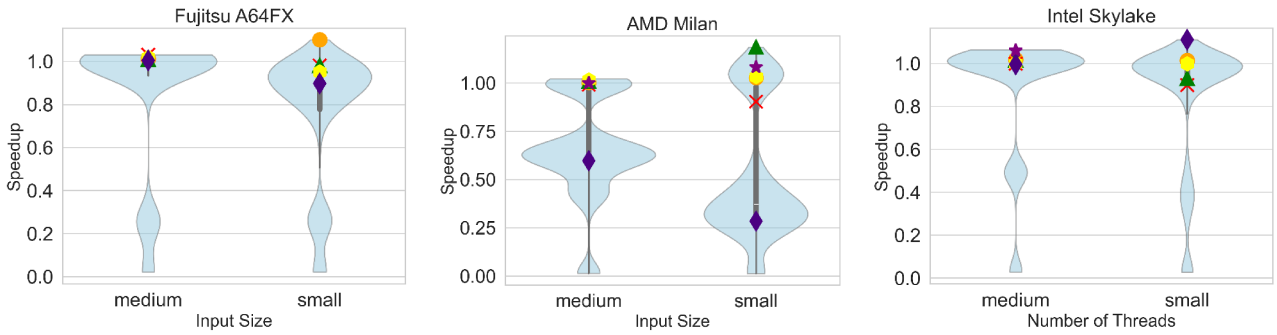

Fig. 1: Performance observed during full search space exploration of the environment variables on the **Alignment benchmark** [2] with different processors. The points marked in all figures refer to the configuration that yielded the best performance in one (architecture and input size) setting and the corresponding location of the same on other settings.

Our analysis and results include a qualitative approach to defining the influence of these variables on application runtime. This is achieved by first collecting performance data with exhaustive runs of variable-value pairs on popular benchmark suites and applications on different CPU architectures. Exhaustive runs of this capacity can help us understand the underlying search space and guide the methodology to tune and recommend promising environment variables and associated settings for different applications. For instance, we see the distribution of performance data points in Fig. 1 for the *Alignment* benchmark from the BSC OMP Task Suite [2] when the input size is varied on all architectures in consideration. Note that the best performing configurations on one architecture and input size, highlighted by differently colored and shaped marks, are not always top-contenders across all execution environments. Further, all our benchmarks show a speedup potential compared to the default configuration, albeit the default performs very well across the board.

As large-scale data collection is non-trivial, we will make all our raw data, for ARM, AMD X86, and Intel X86, available to the community. Further contributions include:

- Insights on the available environment variables for LLVM/OpenMP and a quantitative analysis of their impact on application runtime across various hardware platforms and benchmarks.
- Open sourcing all of our data collected during large-scale sweeps of the parameter space, visualization of the results, and all tooling used in the process.
- Recommendations of environment variables and values to try based on an architecture and application type.

This paper is structured as follows: Section II compares our study to others that tune OpenMP applications. We explain the effect of all considered LLVM/OpenMP environment variables in Section III, along with their potential values and defaults. In Section IV, we introduce the experimental paradigm adopted in our study and share details of the architectures, applications, data collection, cleaning and validation methodology. Raw results and derived insights are presented in Section V before we conclude in Section VI with final remarks and future work.

II. Related Work

With increasing heterogeneity in computing hardware, techniques utilizing machine learning are gaining popularity. ML algorithms are a natural fit for decision-making problems or generating heuristics when vast amounts of data are available. Some seminal works have leveraged ML at different levels of the software stack to tackle decision making problems and provide adaptive heuristics. Trofin, M, et al. [3] conducted a study that applied machine learning (ML) to replace the heuristic for `inlining-for-size` in the LLVM compiler infrastructure. Their framework enables direct integration of ML algorithms into the compiler through a compiler pass. Another group created an end-to-end pipeline [4] for applying deep learning to learn optimization heuristics by using entire application source code as input data, and showed the effectiveness of their approach on two tasks, namely, predicting optimal mapping for heterogeneous parallelism and GPU thread coarsening factors. VenkataKeerthy, S., et al. [5] worked on learning robust distributed embeddings from LLVM IR to represent source code in continuous space. They also demonstrate the effectiveness of their embeddings on the tasks of predicting optimal mapping for heterogeneous parallelism and determining thread coarsening factors.

Additionally, some studies have leveraged ML techniques to optimize parallel applications commonly developed using the OpenMP programming model. A couple of these works have looked at autotuning OpenMP loops to exploit thread-level parallelism and thread placement strategies. FDT [6] is a framework that adapts the number of threads of OpenMP applications by considering contention for locks and memory bandwidth. LIMO [7] monitors the application at runtime and adapts the execution accordingly. Parcae [8] is a framework that comprises a compiler and runtime system to optimize performance. Popov et. al, [9] demonstrate a practical way to automatically explore the sample space of optimization of configurations consisting of thread- and page-mapping, NUMA degree, degree of parallelism, and inter-region interactions however their method is applied to specific parallel regions. They show energy savings on ARM servers.

Dutta et al. [10] have leveraged representation capabilities of graph neural networks to autotune OpenMP loops. Bolet, G, et. al, [11] have explored global optimization strategies for tuning whole OpenMP programs with comparative analysis of Bayesian Optimization, Particle Swarm Optimization and Covariance Matrix Adaptive Evolution Strategy. Parasyris et. al, [12] develop a record and replay technique in LLVM for OpenMP target offloaded programs. They further show how this can be utilized for per-kernel tuning of performance related parameters pertaining to GPUs such as, number of threads, grid size, launch bounds, etc.

Few works have also looked at optimizing thread-level parallelism from a reduced power usage or energy efficiency perspective. Bolet et. al [11] tune a small set of OpenMP applications on a single CPU architecture. Nornir [13] is a runtime system that monitors the application execution and adjusts the resource configurations (DVFS, number of threads, and thread placement) to satisfy either performance or power consumption requirements. Schwarzrock et. al, [14] and Luan et. al, [15] have both developed their own online approach to thread throttling and CPU frequency tuning with the goal of optimizing energy usage in the form of Energy Delay Product (EDP). Their approaches are useful in cases when one must change the thread count during program execution. OpenMPE [16] is designed for energy management and enables programmers to insert new directives in regions where energy savings can be potentially achieved. This work requires a particular compiler and runtime system. Curtis-Murray, M, et. al, [17] present a user-level library for online adaptation of multithreaded code for power-aware high-performance execution. Further, researchers have employed graph neural networks [18] for power-constrained autotuning of OpenMP loops, demonstrating the potential of advanced machine learning techniques in optimizing parallel computing tasks.

However, most works targeting CPUs focus on the same limited set of control mechanisms, namely standardized ICVs, X86 architectures, and their applications are overwhelmingly using structured parallelism, i.a., parallel loops. What sets our study apart is the scale and breath. We use a diverse set of configuration options, which include implementation-defined environment variables, a large selection of applications with parallel loops as well as task-parallelism, and evaluate the vast configuration space on three HPC-relevant architectures. Due to the size of the explored space, we limit ourselves to per-application configurations. This is not a conceptual requirement but matches practical real-world tuning practices.

III. OpenMP Environment Variables

We focus on exploring the parameter space with more variables than those considered in the literature. These include standard-defined and implementation-defined variables. Hereon, variables can be considered as *features*. We consider and describe the following variables: `OMP_PLACES`, `OMP_PROC_BIND`, `OMP_SCHEDULE`, `KMP_LIBRARY`, `KMP_BLOCKTIME`, `KMP_FORCE_REDUCTION` and `KMP_ALIGN_ALLOC`. It is worth noting that the behavior of `OMP_WAIT_POLICY` is derived from `KMP_BLOCKTIME` and `KMP_LIBRARY`. We therefore exclude `OMP_WAIT_POLICY` in favor of the two `KMP_*` variables in our experiments.

Information pertaining to default values and potential values has not been well articulated by other studies. Further, the default values of certain variables in the LLVM/OpenMP are dependent on other settings. In the following we explain the meaning and potential values for each variable and how the default is computed.

1) `OMP_PLACES`: This variable defines how threads are distributed among *places*. The possible values include `threads`, `cores`, `ll_caches`, `sockets`, `numa_domains`, and "unset". If a place was chosen, the OS will allocate threads to that place, while the default, unset, allows threads to be moved. The *place* `numa_domains` requires the $hwloc$ library to be available to the LLVM/OpenMP runtime. This has been omitted in our current experiments and left for future work. Since we did not evaluate "hyper-threading" CPUs, we also ignored the `threads` value.

2) `OMP_PROC_BIND`: The OpenMP standard defines the following five binding/affinity strategies when a parallel region is encountered – `master` (deprecated, now `primary`), `close`, `spread`, `true`, `false`, and "unset". This value is unset by default which corresponds to `false`, however, if `OMP_PLACES` is set, then the default value is `spread`. If the affinity policy is set to `false`, then the threads are free to move from one *place* to another. The LLVM/OpenMP also respects the `KMP_AFFINITY` variable which has an additional set of values, however, we do not consider those.

3) `OMP_SCHEDULE`: This variable controls the schedule kind and chunk size of worksharing-loop directives. A a chunk size is determined by the runtime if not provided or implied. We consider all choices – `static`, `dynamic`, `guided`, and `auto`, but no chunk sizes. The default value is `static`.

4) `KMP_LIBRARY`: This variable selects the LLVM/OpenMP runtime library execution mode. Possible values are `serial`, `throughput`, and `turnaround`. The default value is `throughput`. We do not consider `serial` since it forces parallel applications to run in a serial manner.

5) `KMP_BLOCKTIME`: This variable specifies the duration, in milliseconds (ms), that a thread may wait after completing a parallel region before entering a sleep state. Setting it to `infinite` prevents the thread from sleeping, whereas 0 forces the thread to sleep immediately. A user may select any whole number from [0, `INT32_MAX`] for this variable. The default value is `200`. However, we only consider the following values for experiments – 0, 200, and `infinite`.

6) `KMP_FORCE_REDUCTION`: This variable, which is currently undocumented, determines how a cross-thread reduction is performed. We consider all possible explicitly options – `tree`, `critical`, and `atomic`. The default value is "unset" and a heuristic will determine the method at runtime. If the number of threads is one, no synchronization is needed and a special code path is used. If the number of threads is between

CPU Architecture	#Cores	#Sockets	#NUMA Nodes	Clock Frequency	Memory Type	Memory Capacity
Fujitsu A64FX	48	-	4	1.8 GHz	HBM	32
Intel Xeon Gold 6148 (Skylake)	40	2	2	2.4 GHz[1]	DDR4	188
AMD EPYC 7643 (Milan)	96	2	8	2.3 GHz[1]	DDR4	251

TABLE I: Hardware configuration used in this work.

two and four, the `critical` method is used. Larger thread counts utilize the `tree` method.

7) KMP_ALIGN_ALLOC: This variable, which is also undocumented, is used to define the memory alignment of internal data structures allocated by the LLVM/OpenMP runtime using the `__kmp_allocate` routine. The default value is the cache line size of the architecture. The A64FX processor has a cache line size of 256 bytes whereas the X86 processors considered here have a cache line size of 64 bytes. Therefore, for the A64FX processor we consider 256 and 512, with 256 as the default, and for the X86 processors, we consider 64, 128, 256 and 512, with 64 as the default.

IV. Methodology

Our study is devoted to finding the parameters that work best for the whole application while minimizing the intrusion during development and deployment. Consequently, configurations are not chosen on a "per-kernel", i.a., parallel region, basis but for the entire run. This does not only reduce the search space considerably, but also reflects the fact that users cannot practically tune and modify each kernel in isolation when an application is set up on a system.

Another key aspect of this study is a ground up approach to analyzing the results i.e., we apply visualization tools and statistical techniques to understand and explain relations among different variables, applications, and architectures. We utilize linear and logistic regression models and discuss their benefits and shortcomings. This analysis, described in Section IV-D, reveals underlying relationships which allows us to characterize the influence environment variables per architecture.

We use and evaluate the performance of LLVM/OpenMP over multiple benchmarks in this study. All applications are compiled with LLVM/Clang 15.0.3[2] compilers. This version is fixed across the `aarch64` and `x86` machines. For our evaluations, we use three different CPU micro-architectures spanning two instruction sets, commonly deployed in HPC systems. Key facts about the CPUs are presented in Table I.

A. Benchmark Applications

The considered benchmarking suites and applications are briefly described here. For all the benchmarks, we use the host OpenMP implementation for our experiments.

[1]Clock frequencies of Intel and AMD processors mentioned here reflect their base frequency. Their true clock frequency is variable and is determined by the CPU frequency governor which is set to performance mode.

[2]We utilized this version at the start of our data collection process. Given the stability of the LLVM/OpenMP host runtime across recent versions, our analysis remains valid and applicable to the latest LLVM release.

1) NAS Parallel Benchmarks: The NAS Parallel Benchmarks (NPB) [19] are a small set of programs designed to help evaluate the performance of parallel supercomputers. The benchmark suite has been extended to include new benchmarks for unstructured adaptive meshes, parallel I/O, multi-zone applications, and computational grids. Problem sizes in NPB are predefined and indicated as different classes. We use the following benchmarks written in C and OpenMP: BT, CG, EP, FT, LU, MG.

2) BSC OpenMP Tasking Suite: The objective of the suite is to provide a collection of applications that allow to test OpenMP tasking implementations [2]. We consider the following applications: Alignment, Health, NQueens, Sort, Strassen.

3) RSBench: RSBench [20] is a mini-app representing a key computational kernel of the Monte Carlo neutron transport algorithm. Specifically, RSBench represents the multipole method of performing continuous energy macroscopic neutron cross section lookups.

4) XSBench: XSBench [21] is a mini-app representing a key computational kernel of the Monte Carlo neutron transport algorithm. Specifically, XSBench represents the continuous energy macroscopic neutron cross section lookup kernel.

5) SU3Bench: The kernel is based on the `mult_su3_nn` SU(3) matrix-matrix multiply routine in the MILC Lattice Quantum Chromodynamics (LQCD) code. Matrix-matrix (and matrix-vector) SU(3) operations are a fundamental building block of LQCD applications.

6) LULESH: LULESH [22] approximates hydrodynamics equations discretely by partitioning the spatial problem domain into a collection of volumetric elements defined by an unstructured hex mesh.

B. Data Collection, Cleansing and Pre-Processing

All applications were compiled with the same version of LLVM/Clang built for respective architectures with `-O3 -march=native -fopenmp` compilation flags and therefore, improvements in runtime, if any, are reported over the optimized binaries. For Fujitsu A64FX, `-march` is replaced with `-mcpu` to instruct the compiler to generate SVE instructions where possible.

The benchmarks were executed multiple times in a cluster environment. Execution was batched in a way that all configurations were explored for a setting iteratively. This decision was made to preserve the relative performance of parameterized environment variables for that setting if multiple such runs were not possible. We decided to vary the number of threads and input sizes for applications, but not simultaneously due to the large search space. The performance data distribution generated in each setting is useful for observing how the

trends emerge and change when these variations in number of threads or input sizes are applied. For instance, the NAS Parallel Benchmarks and BSC OpenMP Task Suite are both configured for varied input size, while keeping thread count constant and remaining proxy applications are configured for varied thread counts with default input size.

The raw data is stored and tabular datasets for each setting are created after extracting and processing. Both, the raw output and processed datasets will be open sourced for use by the community. Afterwards, the datasets are further enriched with the runtime of the default settings discussed in Section III. Speedup is then calculated from the observed runtime of an experiment compared to the default configuration. These data files collectively include over 240,000 unique samples, processed from the raw output of application execution and converted into tabular data files. The distribution of samples is shown in Section IV-B.

Architecture	Applications	#Samples
Fujitsu A64FX	15	53822
AMD Milan	13	99707
Intel Skylake	12	90230

TABLE II: Dataset description.

C. Statistical Significance Of The Collected Data

We conduct a Wilcoxon signed-rank test [23] to evaluate the significance of runtime differences of the same configuration when run multiple times i.e., we want to see the variation in observed results per configuration. The test is conducted in pairs of observations for each configuration. These pairs are referred to as R0, R1 and R2. And we showcase the consistency in performance for all pairs of a configuration. This test is applied because the distribution of runtime data is not normally distributed as we can see in Fig. 1.

As an example, we perform this test for the Alignment benchmark on all three processors. The results are shown in Table III. Results are generated as a *statistic* and corresponding *p-value*. A high *p-value* indicates there is no statistically significant difference in the results observed between those pairs. Thus, the runtime measurements across multiple runs are consistent on that architecture. Low *p-values*, as seen for both X86 processors, indicate that there is a statistically significant difference in the runtime measurements. This implies that there are inconsistencies (aka. noise) in the runtime measurements of the benchmarks on these processors.

Table IV shows that the mean and standard deviation calculated for each execution of the Alignment benchmark on an architecture are similar. The means for runtime measurements on A64FX are the same, with minor differences in the std. deviation. For the Milan and Skylake, the means and standard deviation calculated for all measurements are similar for the respective architecture. This was observed across all benchmarks on the X86 architectures. To mitigate variations in runtime of configurations, we average all runtime measurements per configuration.

Architecture-Benchmark	Pair	Test Stat	p-value
a64fx-alignment-small	R0, R1	1157254.5	0.73
	R1, R2	1161973.0	0.86
	R2, R3	1155559.5	0.72
milan-alignment-small	R0, R1	4095517.0	3.23e-12
	R1, R2	1529843.0	0.0
	R2, R3	1564503.5	0.0
skylake-alignment-small	R0, R1	4555474.0	0.19
	R1, R2	2497324.0	4.19e-154
	R2, R3	2598135.0	1.77e-140

TABLE III: Wilcoxon test results for runtime comparisons across architectures.

Architecture-Application	Runtime Idx	Mean (sec)	Std Dev (sec)
a64fx-alignment-small	Runtime_0	0.131	0.310
	Runtime_1	0.131	0.310
	Runtime_2	0.131	0.311
milan-alignment-small	Runtime_0	0.135	0.308
	Runtime_1	0.109	0.265
	Runtime_2	0.111	0.274
skylake-alignment-small	Runtime_0	0.061	0.140
	Runtime_1	0.062	0.142
	Runtime_2	0.062	0.139

TABLE IV: Runtime statistics for different architectures.

D. Analysis Methodology

The combination of environment variables, application parameters, and underlying architecture create a high-dimensional search space. To extract meaningful insights from such high-dimensional data, we need to reduce the number of dimensions. We have considered linear techniques, namely linear and logistic regression to model the data. Linear techniques offer the benefit of providing a certain level of interpretability in the machine learning models and their results. However, they have limitations that make them less effective for accurately modeling high-dimensional data. In such cases, non-linear techniques are more appropriate.

The distribution of points in Fig. 1 indicates that our data does not satisfy the requirements for fitting a linear regression model. This is experimentally observed with low confidence scores associated with poor model fitting. To address this challenge, we reformulate the problem as a classification task. We further annotate our processed data by labeling each sample and apply a classifier to find a separation boundary in high-dimensional space. The samples are labelled as "optimal" if speedup > 1.01, "sub-optimal" otherwise (corresponding to at least 1% improvement in application performance).

To analyze samples across applications or architectures, we encode applications and architectures as "features" in the data. This encoding is a naive numeric scheme. These features act as placeholders for the underlying characteristics they represent. More robust and sophisticated encoding schemes can be applied to more accurately represent architecture details and application embeddings. However, for the purpose of our experiments, this naive encoding scheme works well as indicated by high model prediction scores.

All analysis scripts are written in Python 3 and we use Pandas and Scikit-Learn to clean, aggregate, and normalize

data where necessary. Default features in all cases include the input size, number of threads, and environment variables under consideration. Additional features are added based on the grouping style explained here:

1) *Per Architecture-Application* – here data includes samples from an application when run on a specific processor. This includes additional features of variation from input sizes/thread counts.
2) *Per Application* – here data includes samples from an application across all architectures, input sizes/thread counts.
3) *Per Architecture* – here data includes samples from all applications run on a specific processor. Additional features include applications, varied input sizes/thread counts.

To gather insights from the machine learning model, we use the coefficients of the learned logistic regression model. These absolute values of the magnitude of coefficients act as hyperparameters that control the separation (classification) boundary. Once we get these magnitudes, we weight normalize them and analyze the result. These are described in the following section.

V. Results

We share the insights gathered from our outlined analysis in the form of research questions, heat maps and tables for the benefit of the reader. Darker shades imply larger influence. We note that these analyses are solely data driven, and do not include profiling and any other performance analysis software.

1) What is the upshot potential for applications and does this translate to other architectures?

The observed improvement in runtime expressed as speedup varies from 1.0x to 4.85x across all data collected in this study. A64FX shows this wide range in highest observed speedup from 1.0x to 4.85x with a median improvement of 1.02x. On the Milan architecture, the observed highest speedup ranges from 1.011x to 2.6x with a median improvement of 1.15x. Range of highest speedup on Skylake is from 1.0x to 3.47x with a median of 1.065x.

The same application running on different architectures can have different upshot potential. For instance, in Table V, we show the range of maximum speedup for the Alignment and XSBench benchmarks. The latter only improves minimally on A64FX and Skylake platforms, whereas on Milan more than 2.6x can be achieved. The Alignment benchmark shows consistent potential across architectures.

Application	Architecture	Speedup Range (x)
Alignment	A64FX	1.032 - 1.101
	Milan	1.022 - 1.186
	Skylake	1.065 - 1.111
XSBench	A64FX	1.004 - 1.015
	Milan	1.016 - 2.602
	Skylake	1.001 - 1.002

TABLE V: Speedup range for different applications on different architectures.

The range of speedup observed per application across architectures can be found in Table VI. Here, the range represents the highest upshot potential observed over the default for each architecture.

Application	Speedup Range (x)
Alignment	1.022 - 1.186
BT	1.027 - 1.185
CG	1.000 - 1.857
EP	1.000 - 1.090
FT	1.010 - 1.545
Health	1.282 - 2.218
LU	1.020 - 1.121
LULESH	1.004 - 1.062
MG	1.011 - 2.167
Nqueens	2.342 - 4.851
RSBench	1.004 - 1.213
Sort	1.174 - 1.180
Strassen	1.023 - 1.025
SU3Bench	1.002 - 2.279
XSBench	1.001 - 2.602

TABLE VI: Speedup range for different applications.

2) Does the same set of environment variables define this upshot across architectures for an application?

In our experiments we find that the same set of environment variables may result in speedups across architectures. The performance of an application program can be characterized by the underlying architecture, program representation, input size and number of threads along with the environment variables chosen for this study. The results in Fig. 2, summarize our findings. Columns names "Architecture" and "Input Size" are added to reflect varying architectures and inputs corresponding to the data used for grouping in this analysis.

The deeper colours imply that those features are responsible for explaining the runtime performance more than the others. The architecture column includes the fraction of influence accounted for that application (row). We observe that many applications exhibit some degree of reliance on the architecture, indicating that the same environment variable-value pairs were not consistently beneficial for a given application across all architectures in our study. We also note that the applications from BSC OMP Task Suite show very low reliance on the architecture suggesting that tuning the environment variables once is sufficient to obtain good performance on all architectures.

Note that Sort and Strassen benchmarks show no reliance because they were not executed on the Skylake and Milan processors due to higher traffic on the cluster.

3) Are there any specific variables that work best for an architecture?

Fig. 3 highlights the variables that are generally influential in determining the sway in performance on an architecture. In decreasing order, `OMP_NUM_THREADS`, `OMP_PROC_BIND` and `OMP_PLACES` influence runtime across all considered architectures for the applications we have chosen. This result validates other studies which have primarily focused on tuning these variables or corresponding ICVs.

There is some impact of `KMP_LIBRARY` and `KMP_BLOCKTIME` on performance on all architectures.

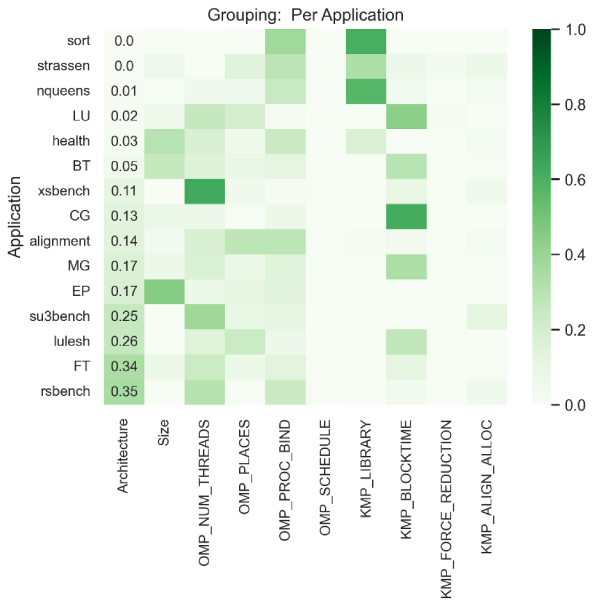

Fig. 2: Heat map highlighting the influence of features when data is grouped by application.

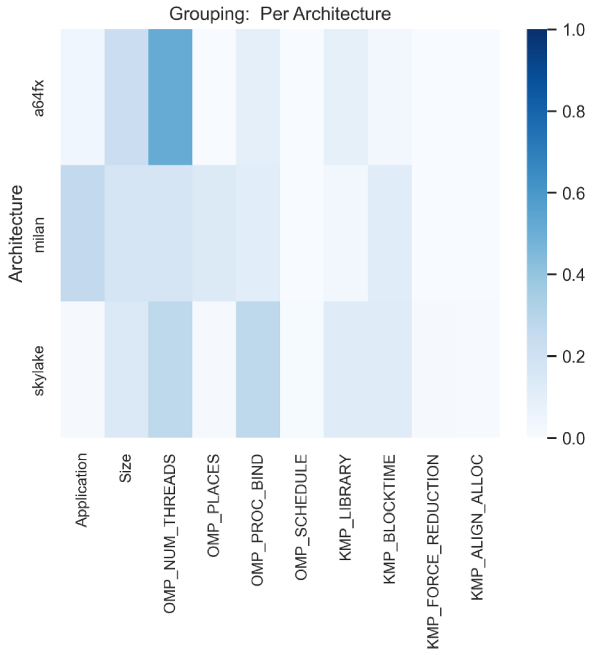

Fig. 3: Heat map highlighting the influence of features when data is grouped by architecture.

Since, `OMP_WAIT_POLICY` is derived from both in LLVM/OpenMP, one may choose to optionally only tune this variable instead of the corresponding `KMP_*` variables. We see very low relevance of `KMP_FORCE_REDUCTION` and `KMP_ALIGN_ALLOC` when applications are grouped by the underlying architecture. However, some reliance is observed in the per-application grouping strategy in Fig. 4.

For an architecture if the "Application" column shows darker shades in Fig. 3, one may choose to look at the next figure describing the impact seen across applications in Fig. 2. Now, if the "Architecture column" has a darker shade in Fig. 2, one may look at the finer grouping of Per Architecture-Application in Fig. 4. This is the hierarchical style of modeling we have adopted here. A user may choose to look at the impact of environment variables at different levels of granularity to identify the most impactful ones to tune for their specific application kind and architecture.

4) What trends (if any) are associated with the worst performance?

A commonly observed trend related to the worst performance is observed with the specific thread placement on cores and threads bound to the master/primary thread when the application was run with large number of threads. This behaviour is expected because the master/primary binding policy will bind the threads to the same "place" as the master/primary thread. In the OpenMP fork-join execution model, the application is executed by the primary thread, and multiple threads are forked when a parallel region is encountered. Based on the master binding policy, these new forked threads will also be located on the same "place" as the primary thread which is not desirable with large number of threads and we recommend this pair of settings to be avoided.

App	Arch	Variable	Value
Nqueens	All	KMP_LIBRARY	turnaround
CG*	A64FX	defaults	defaults
	Milan	KMP_BLOCKTIME	200
		OMP_PROC_BIND	spread/true
	Skylake	KMP_BLOCKTIME	200
		OMP_PLACES	cores/sockets/ ll_caches
		KMP_FORCE _REDUCTION	tree/atomic

TABLE VII: Best Performing Environment Variables and Values.

The high variance in distribution of performance data as portrayed by violin plots, and non-uniformity in factors impacting application performance as shown by the heat maps, makes drawing conclusions and stating which parameters and associated values are better performing very challenging. For example, we share the most impactful performing variables and values for two applications that in Table VII. On one hand, `KMP_LIBRARY=turnaround` significantly improves the NQueen applications performance across all architectures,

whereas we see a different set of variables and associated values affecting the performance of the CG application on different architectures. Therefore, we direct the reader to the heat maps to understand the impact of the additional factors to performance such as architecture, input sizes and thread counts before selecting parameters and respective values.

VI. Conclusion and Future Work

We have performed large scale parameter space exploration of a wide variety of applications on three different CPU architectures. This effort has led to the collection of over 240,000 data samples that have been cleaned and transformed into tabular data files. The distribution of performance data is visualized using violin plots. Initial observations reveal a wide range of performance variations and a non-normal distribution of the data. Additional violin plots displaying the distribution of performance data have been included in the appendix for the reader's reference.

Linear modeling techniques have been specifically chosen for their interpretability aspect. To circumvent inherent restrictions within linear models, we show how data analysis was performed with a surrogate task of classification. This "simplest-first" approach to modeling data is useful in understanding when different variables influence application performance. Fig. 2 and 3 share the features found to be most impactful for both architecture and application grouping strategies respectively.

The results from qualitatively defining the impact of variables can serve as a search space pruning technique. As we have seen, not all environment variables contribute equally to application performance. Therefore, tuning a subset of environment variables can help achieve near optimal performance. The impact of an environment variable is further subject to the application and underlying CPU architecture. Therefore, tuning a subset of variables for that architecture or application can be less expensive to the user.

The outcome of the performed analysis above can also be used in other autotuning studies that aim to find near-optimal configurations by applying discrete search space traversal algorithms. For example, hill climbing algorithms vary the parameter value of one variable at a time while keeping others fixed till all have been parsed. While randomizing the order of variable settings reduces the likelihood of encountering local minima, having information on the impact of variables can further decrease this probability, especially when the dependency relationships between parameters are unclear.

We acknowledge the limitation of this study. All of these analyses were made possible due to the large scale exploration. Given the length of this effort in terms of time, this approach is not scalable from a user's perspective. Another limitation includes the reduced exploration of thread counts for the applications and architectures in consideration. We will add more thread counts and latest CPU chips in the data collection strategy. Given the importance of thread counts, we direct the user to other studies that can recommend thread counts given an application and architecture.

These results can be adopted by users if their OpenMP applications have similar computation patterns to the benchmark applications here. However, this isn't always the case. By viewing the heat maps in Fig. 2 and 3, we see that there is no clear winner for an application or an architecture. Therefore, there is no guarantee this knowledge can be transferred to new *unseen* applications or architectures. However, our methodology can guide future studies and help gain insights in different kinds of applications. The development of non-linear approaches to model such data and devising methods to fine-tune these models with limited data of prior unseen applications is a suitable path forward.

Acknowledgment

The authors thank Jonathan L. Peyton at Intel for sharing useful insights related to the default behaviour of the LLVM/OpenMP Runtime. This material is based upon work by the National Science Foundation under grant no. CCF-2113996. The authors would like to thank Stony Brook Research Computing and Cyberinfrastructure, and the Institute for Advanced Computational Science at Stony Brook University for access to the SeaWulf computing system, which was made possible by a $1.4M National Science Foundation grant (#1531492). The authors would like to thank Stony Brook Research Computing and Cyberinfrastructure, and the Institute for Advanced Computational Science at Stony Brook University for access to the innovative high-performance Ookami computing system, which was made possible by a $5M National Science Foundation grant (#1927880).

References

[1] B. R. de Supinski, T. R. W. Scogland, A. Duran, M. Klemm, S. M. Bellido, S. L. Olivier, C. Terboven, and T. G. Mattson, "The ongoing evolution of openmp," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 2004–2019, 2018.

[2] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade, "Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp," in *2009 International Conference on Parallel Processing*, 2009, pp. 124–131.

[3] M. Trofin, Y. Qian, E. Brevdo, Z. Lin, K. Choromanski, and D. Li, "Mlgo: a machine learning guided compiler optimizations framework," *arXiv preprint arXiv:2101.04808*, 2021.

[4] C. Cummins, B. Wasti, J. Guo, B. Cui, J. Ansel, S. Gomez, S. Jain, J. Liu, O. Teytaud, B. Steiner, Y. Tian, and H. Leather, "Compilergym: Robust, performant compiler optimization environments for ai research," in *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2022, pp. 92–105.

[5] S. VenkataKeerthy, R. Aggarwal, S. Jain, M. S. Desarkar, R. Upadrasta, and Y. N. Srikant, "Ir2vec: Llvm ir based scalable program embeddings," *ACM Trans. Archit. Code Optim.*, vol. 17, no. 4, dec 2020. [Online]. Available: https://doi.org/10.1145/3418463

[6] M. A. Suleman, M. K. Qureshi, and Y. N. Patt, "Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on cmps," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIII. New York, NY, USA: Association for Computing Machinery, 2008, p. 277–286. [Online]. Available: https://doi.org/10.1145/1346281.1346317

[7] G. Chadha, S. Mahlke, and S. Narayanasamy, "When less is more (limo):controlled parallelism forimproved efficiency," in *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ser. CASES '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 141–150. [Online]. Available: https://doi.org/10.1145/2380403.2380431

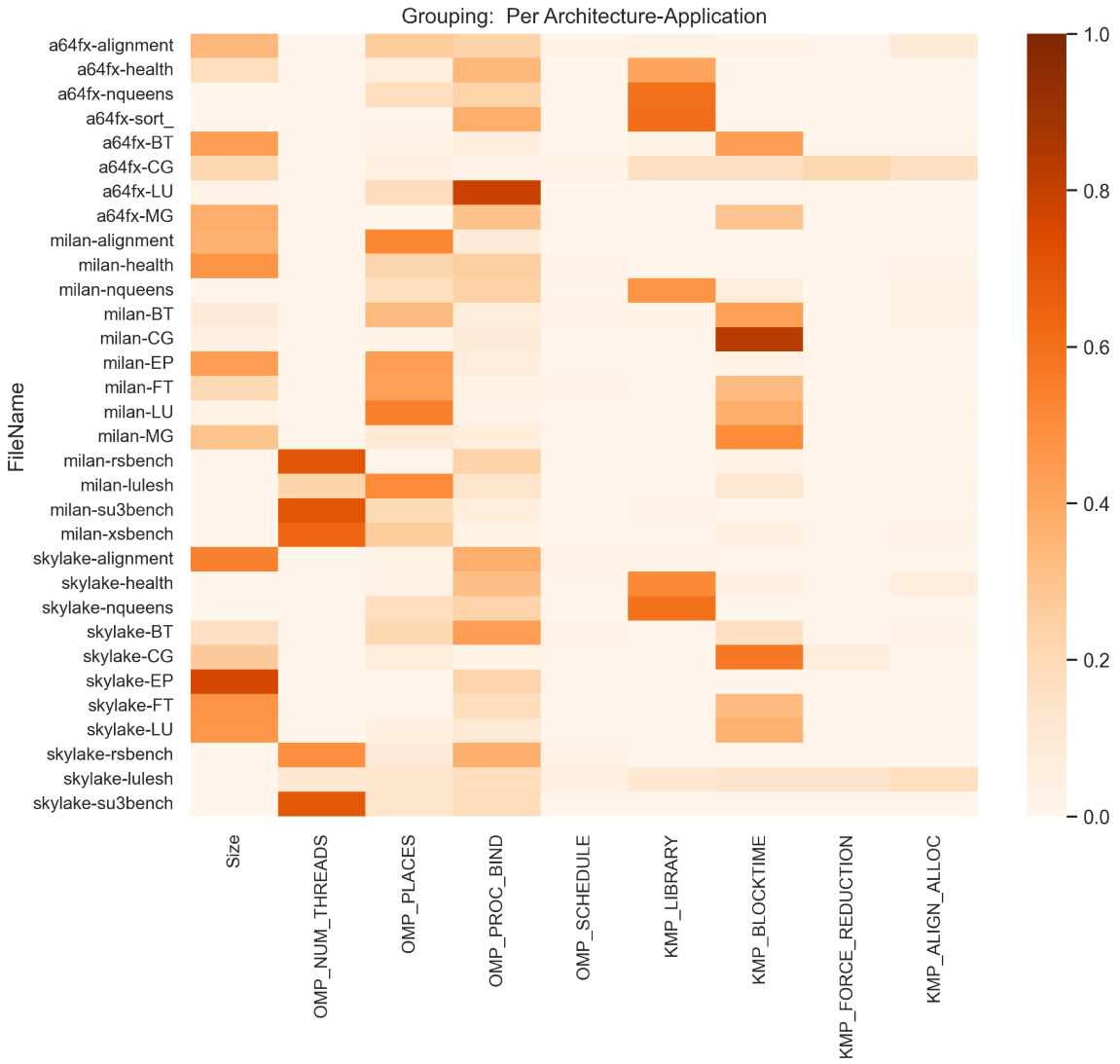

Fig. 4: Heat map highlighting the influence of features when data is grouped by application-architecture.

[8] A. Raman, A. Zaks, J. W. Lee, and D. I. August, "Parcae: a system for flexible parallel execution," *SIGPLAN Not.*, vol. 47, no. 6, p. 133–144, jun 2012. [Online]. Available: https://doi.org/10.1145/2345156.2254082

[9] M. Popov, A. Jimborean, and D. Black-Schaffer, "Efficient thread/page/parallelism autotuning for numa systems," in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 342–353. [Online]. Available: https://doi.org/10.1145/3330345.3330376

[10] A. Dutta, J. Alcaraz, A. TehraniJamsaz, A. Sikora, E. Cesar, and A. Jannesari, "Pattern-based autotuning of openmp loops using graph neural networks," in *2022 IEEE/ACM International Workshop on Artificial Intelligence and Machine Learning for Scientific Applications (AI4S)*. Los Alamitos, CA, USA: IEEE Computer Society, nov 2022, pp. 26–31. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/AI4S56813.2022.00010

[11] G. Bolet, G. Georgakoudis, K. Parasyris, K. W. Cameron, D. Beckingsale, and T. Gamblin, "An exploration of global optimization strategies for autotuning openmp-based codes," in *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2024, pp. 741–750.

[12] K. Parasyris, G. Georgakoudis, E. Rangel, I. Laguna, and J. Doerfert, "Scalable tuning of (openmp) GPU applications via kernel record and replay," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2023, Denver, CO, USA, November 12-17, 2023*, D. Arnold, R. M. Badia, and K. M. Mohror, Eds. ACM, 2023, pp. 28:1–28:14. [Online]. Available: https://doi.org/10.1145/3581784.3607098

[13] D. De Sensi, M. Torquati, and M. Danelutto, "A reconfiguration algorithm for power-aware parallel applications," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 4, dec 2016. [Online]. Available: https://doi.org/10.1145/3004054

[14] J. Schwarzrock, C. C. de Oliveira, M. Ritt, A. F. Lorenzon, and A. C. S. Beck, "A runtime and non-intrusive approach to optimize edp by tuning threads and cpu frequency for openmp applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 7, pp. 1713–1724, 2021.

[15] G. Luan, P. Pang, Q. Chen, S. Xue, Z. Song, and M. Guo, "Online thread auto-tuning for performance improvement and resource saving," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 3746–3759, 2022.

[16] F. Alessi, P. Thoman, G. Georgakoudis, T. Fahringer, and D. S.

Nikolopoulos, "Application-level energy awareness for openmp," in *OpenMP: Heterogenous Execution and Data Movements*, C. Terboven, B. R. de Supinski, P. Reble, B. M. Chapman, and M. S. Müller, Eds. Cham: Springer International Publishing, 2015, pp. 219–232.

[17] M. Curtis-Maury, J. Dzierwa, C. D. Antonopoulos, and D. S. Nikolopoulos, "Online power-performance adaptation of multithreaded programs using hardware event-based prediction," in *Proceedings of the 20th Annual International Conference on Supercomputing*, ser. ICS '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 157–166. [Online]. Available: https://doi.org/10.1145/1183401.1183426

[18] A. Dutta, J. Choi, and A. Jannesari, "Power constrained autotuning using graph neural networks," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2023, pp. 535–545. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/IPDPS54959.2023.00060

[19] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The nas parallel benchmarks—summary and preliminary results," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '91. New York, NY, USA: Association for Computing Machinery, 1991, p. 158–165. [Online]. Available: https://doi.org/10.1145/125826.125925

[20] J. R. Tramm, A. R. Siegel, B. Forget, and C. Josey, "Performance analysis of a reduced data movement algorithm for neutron cross section data in monte carlo simulations," in *Solving Software Challenges for Exascale*, S. Markidis and E. Laure, Eds. Cham: Springer International Publishing, 2015, pp. 39–56.

[21] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz, "XSBench - the development and verification of a performance abstraction for Monte Carlo reactor analysis," in *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*, Kyoto, 2014. [Online]. Available: https://www.mcs.anl.gov/papers/P5064-0114.pdf

[22] I. Karlin, J. Keasler, and R. Neely, "Lulesh 2.0 updates and changes," Lawrence Livermore National Laboratory, Tech. Rep. LLNL-TR-641973, August 2013.

[23] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945. [Online]. Available: http://www.jstor.org/stable/3001968

APPENDIX

We share more violin plots of the performance data distribution for the reader's benefit in Fig. 5 to 7.

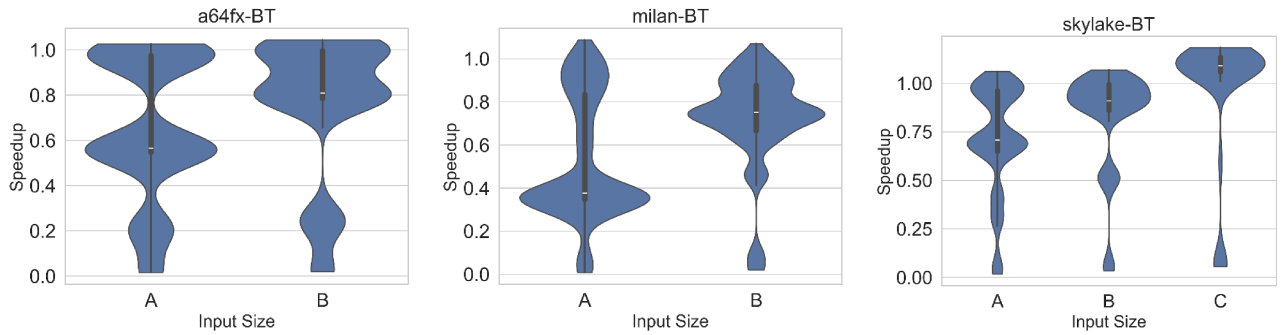

Fig. 5: Performance observed during full search space exploration of the Environment Variables on the **BT benchmark** [19] with different processors.

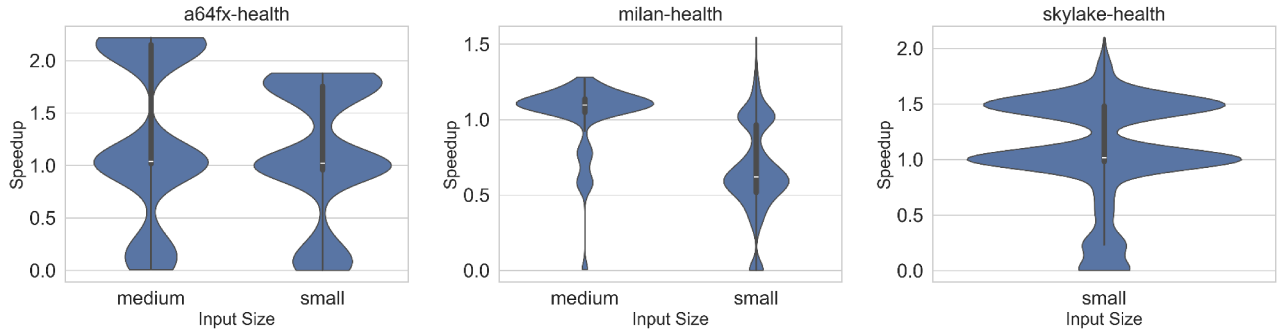

Fig. 6: Performance observed during full search space exploration of the Environment Variables on the **Health benchmark** [2] with different processors.

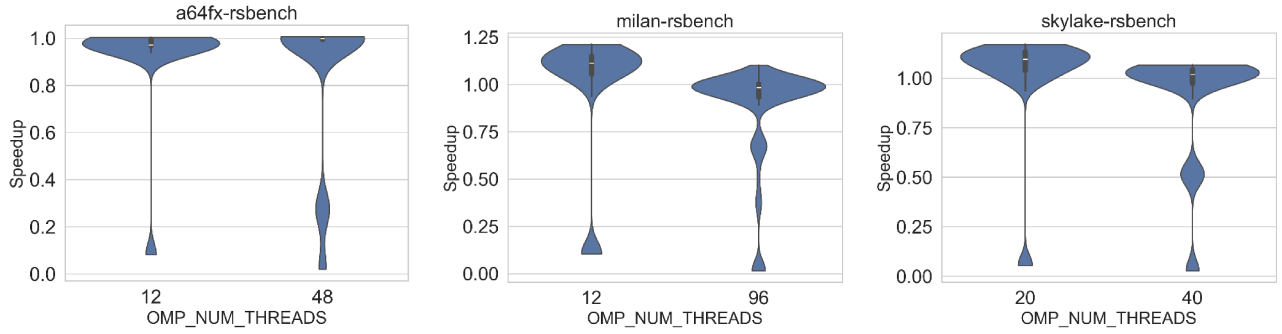

Fig. 7: Performance observed during full search space exploration of the Environment Variables on the **RSBench proxy application** [20] with different processors.