Off-Road Performance Modeling – How to Deal with Segmented Data

M. Kashif Ilyas, Alexandru Calotoiu, and Felix Wolf

Technische Universität Darmstadt, 64289 Darmstadt, Germany
cashif.pk@gmail.com, {calotoiu,wolf}@cs.tu-darmstadt.de

Abstract. Besides correctness, scalability is one of the top priorities of parallel programmers. With manual analytical performance modeling often being too laborious, developers increasingly resort to empirical performance modeling as a viable alternative, which learns performance models from a limited amount of performance measurements. Although powerful automatic techniques exist for this purpose, they usually struggle with the situation where performance data representing two or more different phenomena are conflated into a single performance model. This not only generates an inaccurate model for the given data, but can also either fail to point out existing scalability issues or create the appearance of such issues when none are present. In this paper, we present an algorithm to detect segmentation in a sequence of performance measurements and estimate the point where the behavior changes. Our method correctly identified segmentation in more than 80% of 5.2 million synthetic tests and confirmed expected segmentation in three application case studies.

Keywords: parallel computing, performance tools, performance modeling

1 Introduction

The increasing number of processors in our computing hardware poses new challenges to developers. Increasing software parallelism challenges traditional ways of writing and debugging programs. Badly designed parallel programs may fail to reach the expected performance when run on a larger number of processors. Therefore, finding and removing scalability bugs is key to the achievement of sustainable parallel performance. The term scalability bug refers to those parts of a program whose scaling behavior is unintentionally poor, i.e., which perform worse than expected when using a larger number of processors [2]. As scalability bugs do not become manifest unless the program is actually run at larger scales, it is very difficult for developers to discover them. Often, they are found when the software is already deployed and changes are more expensive.

One approach capable of finding such bugs early and easily is empirical performance modeling: a performance model of a program is built from measurements of relevant performance metrics. We can do this even for individual regions

of the code, henceforth called kernels. Typically, we run the program on different numbers of processors p and and measure the metric m of interest for each run, creating for each kernel data points of the form (p, m). These data points are then analyzed using regression and turned into a mathematical performance model of the kernel. Empirical models are not necessarily as accurate as analytical models but are good enough to show the scaling trend of the kernel. Problematic kernels can then be examined by the developer in more detail. The whole process can be automated to obtain empirical models for all possible kernels of a program, hence avoiding the risk of overlooking any critical kernel, at least for the given input set.

Extra-P [2] is an automatic tool that implements the above approach. It generates empirical models for each kernel (i.e., call path) of a program in a human-readable form. Extra-P also extrapolates performance to a chosen target scale such that it can be compared with developer expectations. While Extra-P's workflow is quite effective in finding scalability bugs, it fails if the input data represents two or more distinct behaviors of a program. Extra-P assumes that the performance of a kernel can be characterized by a single function, however, some kernels do not follow a single trend in every situation. There are many practical scenarios where programs change their behavior. For example, modern MPI implementations switch from one algorithm to another, depending on the message size, the number of processes, or the network topology [9]. Overlooking such segmentation not only results in the creation of inaccurate models but also poses the risk of ignoring potential scalability bugs or confusing the user with false positives.

In this paper, we introduce a novel method to detect such segmentation before generating empirical models. Driven by the requirements of performance modeling in HPC, where trial runs can be quite expensive, a particular challenge our method addresses is the low number of data points. Specifically, we propose

- an algorithm to find segmentation in data with as few as six points, and
- a method to estimate the change point.

Our approach (i) correctly identified segmentation in more than 80% of more than five million randomly generated datasets and (ii) confirmed expected segmented behavior in three realistic use cases, including a climate code, a simple matrix multiplication benchmark, and several MPI collective operations.

In the next section, we review the existing workflow of Extra-P and explain how it struggles with segmented data. In Section 3, we explain our approach with the help of an example. We demonstrate its effectiveness in Section 4. In Section 5, we compare it to related work and argue why it fits our purpose best. We conclude the paper in Section 6, where we also discuss future work.

2 Performance Modeling with Extra-P

As our work is intended to improve Extra-P, we briefly review how it generates models from performance data. Extra-P exploits the observation that performance models of most practical programs can be expressed as n terms involving

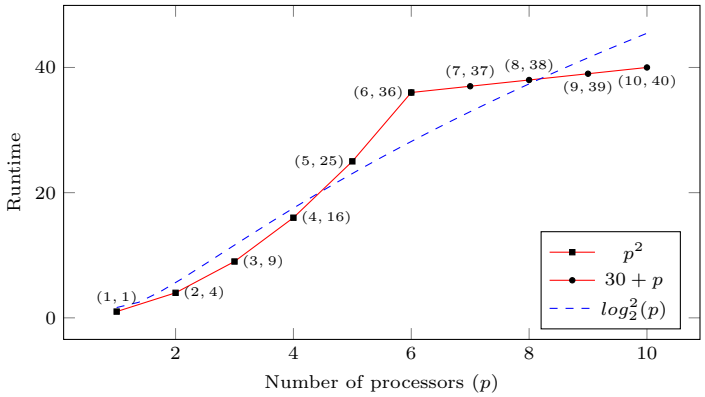


Fig. 1: Data points from two different functions (solid lines) and the model generated by Extra-P (dashed line).

logarithms and powers of the model parameter p, which is usually the number of processors but can also be something different like the input size. Hence, performance models can be represented in the *performance model normal form* (PMNF):

$$f(p) = \sum_{k=1}^{n} c_k \cdot p^{i_k} \cdot log_2^{j_k}(p)$$

As identification of scalability bugs rather than prediction accuracy is the primary objective of Extra-P, the sets I and J from which i_k and j_k are selected, do not need to be arbitrarily large to obtain reasonably accurate models. The authors suggest $n = 2$, $I = \left\{\frac{0}{2}, \frac{1}{2}, \frac{2}{2}, \frac{3}{2}, \frac{4}{2}, \frac{5}{2}, \frac{6}{2}\right\}$ and $J = \{0, 1, 2\}$ as default. The generation algorithm starts with a set of simple (i.e., short) candidate models and chooses the winner using K-fold validation. The size of the candidates is gradually increased until either the maximum of n is reached or signs of overfitting appear.

This approach works as long as all the performance measurements represent a single behavior. However, if a certain kernel exhibits segmented behavior, i.e., its performance trend changes after a certain point, the resulting model will be inaccurate. To clarify our point, we consider an example of segmented data and the corresponding model generated by Extra-P, which are shown in Figure 1. The data was generated using the functions $f_1(p) = p^2$ for $p \in \{1, 2, ..., 5\}$ and $f_2(p) = 30 + p$ for $p \in \{6, 7, ..., 10\}$. With these data as input, Extra-P generates the model $f(p) = 1.65 + 3.97 \cdot log_2^2(p)$. This is misleading because the actual functions are quadratic and linear, but not logarithmic. At $p = 1024$, the prediction error of the model is almost 62%.

3 Approach

Our algorithm is designed to help Extra-P detect segmentation in the data before models are generated. Its input is a set of performance measurements, while the

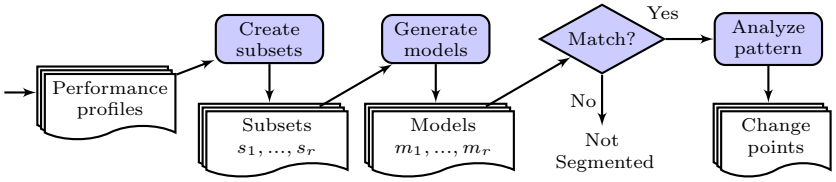


Fig. 2: Steps involved in segematation detection and change-point identification.

output indicates whether the given measurements show segmented behavior or not. If the data turns out to be segmented, the algorithm tries to identify the change point. With this information, Extra-P can generate separate models for each segment and/or request new measurements if any segment is too small for model generation. Figure 2 highlights the different steps of our algorithm. Below, we discuss each step in detail and apply it to the example from Section 2.

3.1 Detecting Segmentation

Our algorithm rests on the observation that models generated from *homogeneous* subsets of the input data set (i.e., subsets representing a single behavior), will differ from models created from *heterogeneous* subsets (i.e., subsets representing multiple behaviors). In this work we focus on subsets defined by consecutive data points corresponding to subintervals of the input parameter.

As an example, we try to divide the segmented data from Figure 1 into subsets representing five consecutive values of the model parameter p. We use a sliding window of five measurement points to create subsets which results in a total of six subsets, each containing five points. The first subset s_1 contains the first five points $\{1, 4, 9, 16, 25\}$, the second subset s_2 contains the five points starting from second point $\{4, 9, 16, 25, 36\}$, and so on. All these subsets are listed in Table 1. Note that the subsets s_1, s_2, and s_6 are homogeneous, as they exhibit a single behavior while the subsets s_3, s_4, and s_5 are heterogeneous, mixing points from the two behaviors. Then, we create models $m_1, \ldots, m_r$ for each of the subsets $s_1, \ldots, s_r$, respectively. The number of subsets r depends on the number of input data points and on the number of values each subset is allowed to contain. Ideally, each subset should contain five data points, as recommended in the original research work by Calotoiu et. al [2]. Below, we introduce two model properties that can be used to decide whether a subset is segmented or not.

Absolute nRSS. We define a generalized error value for each model, which is a normalized form of the common RSS. Residual Sum of Squares (RSS) is a measure of the discrepancy between the data and an estimation model and is used to measure the goodness of a model. The normalized residual sum of squares (nRSS) is calculated by dividing the square root of the RSS by the mean of the

Table 1: Subsets created for the data from Figure 1, their respective models, and their nRSS values. Heterogeneous subsets are highlighted.

Subset	Model	nRSS	ϵ
$s_1 = \{1, 4, 9, 16, 25\}$	p^2	≈ 0	–
$s_2 = \{4, 9, 16, 25, 36\}$	p^2	≈ 0	≈ 1
$s_3 = \{9, 16, 25, 36, 37\}$	$-49.41 + 33.45 \cdot \sqrt{p}$	0.18	$5 \cdot 10^{18}$
$s_4 = \{16, 25, 36, 37, 38\}$	$-28.53 + 23.17 \cdot log_2(p)$	0.19	1.05
$s_5 = \{25, 36, 37, 38, 39\}$	$-6.19 + 14.83 \cdot log_2(p)$	0.16	0.84
$s_6 = \{36, 37, 38, 39, 40\}$	$30 + p$	≈ 0	≈ 0

points used to generate the models:

$$nRSS = \frac{\sqrt{RSS}}{\bar{y}}$$

Calculating the nRSS for each subset yields r error terms $e_1, \ldots, e_r$. For our example data from Figure 1, we get six models and their corresponding nRSS values, which are shown in Table 1. The nRSS of the heterogeneous subsets is much higher than the one of the homogeneous subsets because a well-fitting model cannot be found for such diverse data. We identify a subset s_i as potentially heterogeneous if its nRSS $e_i > 0.1$ and homogeneous otherwise. We classify a data set as segmented if the maximum absolute value of the nRSS across all subsets exceeds a threshold of $\theta = 0.5$, an empirically determined value reflecting our experiences after analyzing more than five million synthetic data sets. In most cases, using $\theta = 0.5$ correctly identifies segmented behavior if it exists, while producing only few false positives (i.e., non-segmented behavior falsely identified as segmented).

Relative nRSS. The secondary indicator is the relative nRSS, which is the ratio of the nRSS values of two consecutive subsets. It is applied only when $0.1 \leq nRSS \leq 0.5$. The relative nRSS ϵ can be mathematically expressed as $\epsilon_i = e_{i+1}/(e_i + \eta)$. η is a minimal non-zero value added to avoid division by zero. This criterion has the advantage that it rules out false positives that occur when noise lifts all errors above the threshold. It also covers those scenarios where the absolute nRSS values are smaller than the threshold but the heterogeneous subsets still show a much higher nRSS than the homogeneous ones. We found that $\epsilon > 4$ provides a good additional criterion to determine segmentation when $0.1 \leq nRSS \leq 0.5$.

In our example from Table 1, it is clear that the heterogeneous subsets s_3, s_4 and s_5 have much higher absolute nRSS values than the homogeneous ones, but the maximum resides still below the threshold. However, because $\epsilon_3 >> 4$ we still conclude correctly that the data is segmented.

3.2 Identifying the Change Point

Identifying the change point goes beyond a binary decision whether the data is segmented or not. If a change point can be detected, then a separate model for

Fig. 3: Selection of points for the subsets $s3$, $s4$ and $s5$. Squares and circles represent data points from different behaviors, the sixth point is common to both behaviors.

each behavior, divided by the change point, can be determined, provided enough data points are available. To accomplish this, we tag each subset with a zero if its nRSS classifies it as homogeneous and with a one otherwise. For the data from Table 1, we obtain the pattern 001110.

For the sake of simplicity, we assume that there is a single change point in the data, but the same method can be extended to multiple change points via recursion. Since we create subsets containing five points and we assume that only two different behaviors are present in the data, at most four subsets can be heterogeneous (one point representing the first behavior combined with four points representing the second one, then two combined with three, and so forth). If the two behaviors share a common data point such as in the example, only three such subsets will exist. Therefore, each sequence of values corresponding to the series of subsets will contain either three or four ones, preceded and followed by an arbitrary number of zeros.

The location of the change point can therefore be deduced by examining only the pattern of ones. Practically, we select the subset corresponding to the second one in the pattern. If a common data point for both behaviors exists and therefore the pattern contains three ones, then the change point will be the third data point of that subset. If no common data point for both behaviors exists, then the change point will be between the third and the fourth data point of that subset. In the example, the relevant subsets are $s3$, $s4$, and $s5$, thus the change point is $p = 6$, as shown in Figure 3. In most cases, we do not see a single change point, but two points between which the change happens.

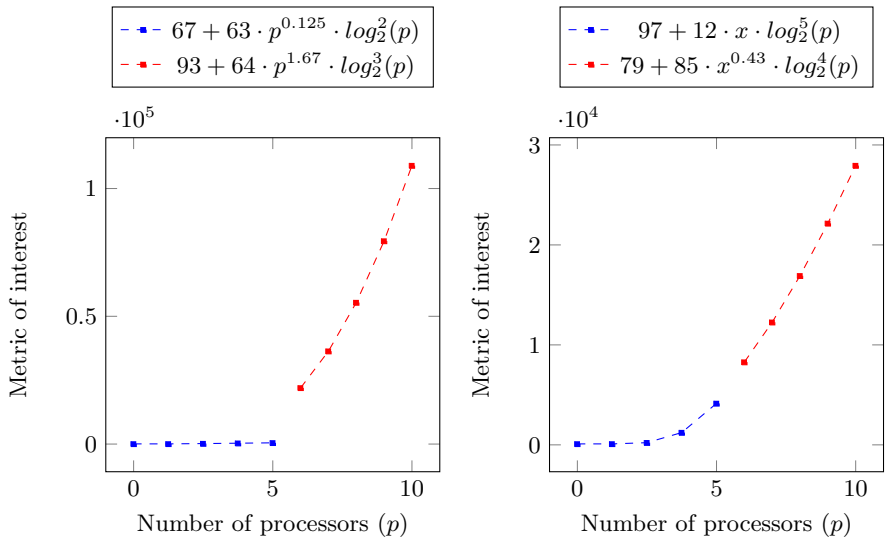
4 Evaluation

To ensure our method correctly distinguishes segmented from non-segmented behavior, we first applied it to millions of synthetic data sets. After that, we tested it with application data known to be segmented, which we correctly identified as such without producing false positives.

4.1 Synthetic Data

We ran our algorithm on data from two categories of randomly generated functions:

– Functions guaranteed to be within the search space, with randomly generated coefficients and exponents chosen at random from those present in the search



(a) True positive

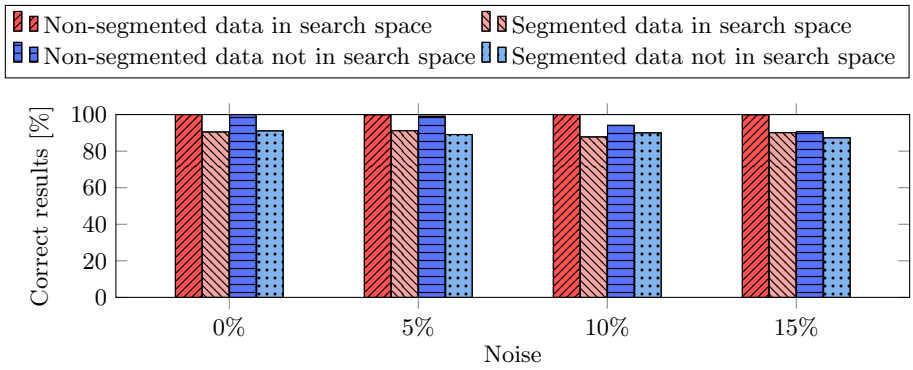
(b) False negative

Fig. 4: Two examples from synthetic data set used for evaluation of the algorithm. Both functions were generated with random coefficients and exponents and a 5% of noise was added. In first case, the segmentation was correctly detected while the second case is a false negative because algorithm failed to detect segmentation.

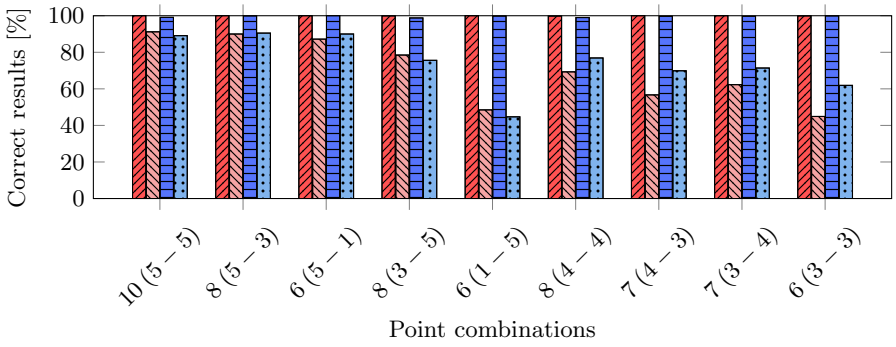
space. These functions can be exactly matched by the algorithm and errors will only appear due to noise or segmentation.

– Functions not guaranteed to be within the search space, with randomly generated exponents and coefficients. These functions are unlikely be matched exactly and likely to have larger overall errors, therefore making the detection of segmented behavior harder.

For each category, we generated tests using data from one function, which should not be marked as segmented, and from two different functions, which should be marked as segmented. A failure in the first case is a false positive, while a failure in the second case is a false negative. Additionally, to observe the accuracy of our algorithms under production conditions, we applied different levels of noise ranging from 0% to 15%. For a noise level of x%, we added a randomly selected percentage of noise between -x and x to the original value. For each noise level, Figure 5a presents the percentages where the algorithm correctly identified the data as segmented or not segmented. The algorithm was always provided with ten data points, either all from one function or equally divided among two different functions. Figure 4a shows an example where randomly generated data was correctly detected as segmented while Figure 4b represents a case of false negative.



(a) Fraction of correct results for different noise levels.



(b) Fraction of correct results for different point combinations with 5% noise. The x-axis shows the total number of points used. The numbers in brackets represent the split of points between the different behaviors.

Fig. 5: Fractions of correctly classified data sets (i.e., neither falsely positive nor falsely negative) for different levels of noise. Each bar was created by analyzing 100,000 synthetic data sets.

Apart from different noise levels, we also tested the accuracy of our algorithm for different numbers and combinations of measurement points. We used a maximum of ten and a minimum of six points and tried various combinations of contributions from each function. In general, our algorithm works best when there are ten or more data points with at least five data points from each function. If there are less than five data points from either of the functions, the percentage of true positives decreases. Figure 5b shows the fractions of point combination where data was correctly identified as segmented or not segmented.

For functions within the search space, the algorithm correctly found the location of the change point in about 90% of the cases. However, the percentage decreases with increasing noise. In those cases where the functions were not guaranteed to be in the search space, the algorithm correctly found the location of the change point around 70% of the time.

Our approach generates less than 1% false positives for a noise level of upto 5%, sparing the user unnecessary confusion and work. With as few as six data points, or one measurement more than usually required by Extra-P, our approach correctly identifies more than 50% of the occurrences of segmented data, and this percentage increases sharply if more measurements are made available. The user can therefore obtain significant gains at very little additional cost.

4.2 Case Studies

In this section, we present three cases studies where we correctly identify expected segmentation in real performance measurements. One of the presented applications, HOMME [4], had already been studied before and has a known segmentation in performance measurements while the other two, namely matrix multiplication and MPI collective operations, are expected to exhibit such a behavior based on how they work.

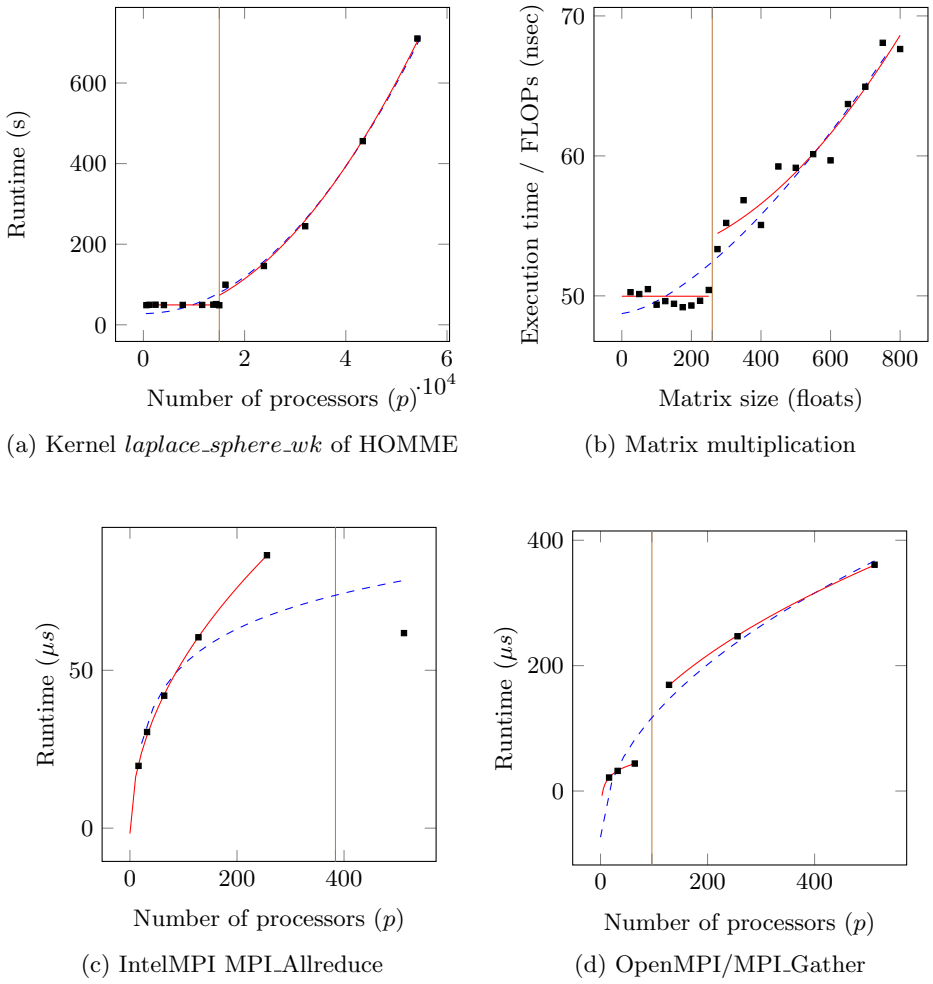
HOMME. This code is the dynamical core of the Community Atmosphere Model (CAM). The scalability of HOMME was studied by Calotoiu et. al [2] and in addition to identifying scalability issues, they found certain kernels to exhibit segmentation. We used performance measurements with processor counts $p \in \{600, 1176, 2400, 4056, 7776, 11616, 13824, 14406, 15000, 16224, 23814, 31974, 43350, 54150\}$ which were taken on the IBM Blue Gene/Q system Juqueen in Jülich according to developer recommendations.

Our algorithm identified 25 out of 664 kernels as segmented and estimated the change point each time to lie between $15,000$ and $16,224$. The execution time of one such kernel, laplace_sphere_wk, was previously characterized as $f(p) = 27.7 + 2.23 \cdot 10^{-7} \cdot p^2$ using the non-segmented algorithm. Using the segmented approach, we came up with the following segmented model:

$$f_{seg}(p) = \begin{cases} 49.36 & p \leq 15,000 \\ 20.8 + 2.3 \cdot 10^{-7} \cdot p^2 & p \geq 16,224 \end{cases}$$

Figure 6a shows the measured execution times and both segmented and non-segmented models for this kernel. The reason for this segmentation is a ceiling term in the code, causing those kernels to be called only once when using 15,000 processors or less, hence resulting in constant time. However, beyond 15,000 processors, the kernels are called quadratically, causing quadratic models to appear. This case study illustrates the advantage of our algorithm, which can detect such segmentation automatically without any user intervention.

Matrix multiplication. A practical scenario of abrupt change in behavior is the effect of cache spilling. The runtime of a memory-bound program heavily depends on the time required to fetch data from the memory. If the data is small enough to fit in the cache, the runtime stays very small. However, as soon as the amount of data exceeds the size of the cache and memory access time becomes the limiting factor, the runtime changes abruptly and follows a completely different pattern.



(a) Kernel *laplace_sphere_wk* of HOMME

(b) Matrix multiplication

(c) IntelMPI MPI_Allreduce

(d) OpenMPI/MPI_Gather

Fig. 6: Graphs showing measurement points, non-segmented models (dashed lines) and segmented models (solid lines). Estimated locations of change points are shown by vertical lines.

We used a sequential naive matrix multiplication to observe this effect and see whether our algorithm can correctly identify the change. We ran our program with increasing matrix sizes and measured the runtime for each matrix size on an Intel Core i5 processor with 2 cores, one 256 kB L2 cache per core, and a 3 MB shared L3 cache. We then divided the runtime by the number of FLOPs to measure the influence of the memory-access time. As the matrix reached a size large enough to not fit in the L2 cache, we saw a sudden drop in the performance (Figure 6b). We not only detected the segmentation of data but also identified the change point between a matrix size of 244 kB and 295 kB, which roughly

Table 2: MPI collective operations which exhibit segmentation. While the left model is listed for all operations, the right model is listed only for those where we had enough data points to create one.

Library	Collective operation	Segmented models	
		Left	Right
IntelMPI	`MPI_Gather`	$4.80 + 0.06 \cdot p \cdot log_2(p)$	Not enough points
	`MPI_Allreduce`	$-1.71 + 5.50 \cdot \sqrt{p}$	Not enough points
OpenMPI	`MPI_Gather`	$-23.3 + 11.17 \cdot log_2(p)$	$-23.11 + 16.95 \cdot \sqrt{p}$

matches the size of the L2 cache. Overall, the segmented model much better reflects the memory hierarchy than the more inaccurate unified model.

MPI collective operations. The performance of MPI collective operations highly depends on the network topology, the number of processes, and the size of messages. Some algorithms perform very well on short messages but fail to perform the same way on larger messages while others behave the opposite way [10]. To maximize performance in every situation, modern MPI libraries offer a wide variety of algorithms for each collective operation and switch between them according to the environment [9,5]. Of course, switching between algorithms leads to segmented performance behavior.

To study this, we measured the runtimes of selected MPI collective operations from Open MPI and Intel MPI for different numbers of processes $p \in \{16, 32, 64, 128, 256, 512\}$ on the Lichtenberg Cluster of TU Darmstadt. We found that MPI_Allreduce and MPI_Gather from Intel MPI change their behavior after 256 and 128 processes, respectively. With Open MPI, such behavior was only noticed for MPI_Gather after 64 processes, as shown in Table 2. The change point was in agreement with the threshold found in the code base of Open MPI, at which the default decision function switches the gather algorithm from linear to binomial. The performance measurements and both the segmented and the non-segmented models are shown in Figures 6c and 6d. Despite restricting the segmentation analysis to only six data points in this case study, our algorithm provides valuable feedback on the performance variability of these collectives. By looking at the results of the analysis, the user can know which extra measurement points he needs to provide to improve model accuracy. It is also evident from Figure 6 that the non-segmented models predicted by Extra-P would result in misleading predictions at higher scales.

5 Related Work

The change point estimation problem has been discussed in the literature since 1966. Several algorithms have been suggested and used by researchers since then and are being further improved to this day. Auger and Lawrance [1] suggested an algorithm in 1989 to find segmented neighborhoods in the data collected by the experiments of molecular biology. The algorithm runs in $O(kn^2)$ time,

where k is the maximum number of change points. The algorithm was later improved by Jackson et. al. [6] to decrease the time complexity to $O(n^2)$. In 2012, Killick et. al [7] proposed a variation of the same algorithm that runs in linear time in the best case, but incurs quadratic cost in the worst case. All the above mentioned methods are collectively called *optimal partitioning* methods and give the exact location of the change point. However, these algorithms are too slow and need much more points than we have in our experiments. Another popular method and the most similar one to ours was suggested by Scott and Knott [8] in 1944 and is known as *binary segmentation*. It is a recursive algorithm that finds the change points by first finding a point and then recursively dividing and reapplying the method to each segment. It continues to do so until no more change points can be found. To tackle multiple change points, our method could be used in a similar way, but in general too few data points are provided to justify its recursive application. The main difference, however, is that our change-point identification scheme is much simpler and faster.

Shao-Tung et. al [3] used fuzzy c-partitioning as a way to find change points in data. They argue that finding change points is similar to arranging data in clusters and hence fuzzy logic can be applied. Similarly, Bingwen et. al [11] used the sparse group lasso (SGL) method to estimate change points. In SGL, two penalty terms of the fitting function make sure to find the best fit for the data. Both of the methods, however, require much more data points than our method to give any reasonable answers. It is important to mention that all of those methods are generic and do not take advantage of the small search space resulting from the performance model normal form like our method does and hence, are much slower, more complicated, and need more data points than our method does.

6 Conclusion

The results of our synthetic data tests as well as the case studies confirm that the proposed algorithm can be used as an effective way to find segmentation in performance data when creating empirical performance models. The suggested algorithm does not require any extra effort on the user's side, is very simple to implement, and can work very well on as few as six points. The algorithm has correctly identified segmented behavior, and did not signal such behavior when none was present in more than 80% of 5.2 million test cases. Hence, it is capable of finding segmentation in the majority of cases, where it would go unnoticed otherwise and leave the user with inaccurate models. We plan to integrate our approach into the next release of Extra-P.

Acknowledgements. This work was supported in part by the German Research Foundation (DFG) through the Priority Programme 1648 *Software for Exascale Computing* (SPPEXA) and the Programme *Performance Engineering for Scientific Software*. Additional support was provided by the German Federal Ministry of Education and Research (BMBF) under Grant No. 01IH16008, and by the

US Department of Energy under Grant No. DE-SC0015524. Finally, we would like to thank the University Computing Center (Hochschulrechenzentrum) of TU Darmstadt for providing us with access to the Lichtenberg Cluster.

References

1. I. Auger and C. Lawrance. Algorithms for the optimal identification of segment neighborhoods. *Bulletin of Mathematical Biology*, 51(1):39–54, February 1989.
2. A. Calotoiu, T. Hoefler, M. Poke, and F. Wolf. Using automated performance modeling to find scalability bugs in complex codes. *SC Í13 Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, November 2013.
3. S. Chang, K. Lu, and M. Yang. Fuzzy change-point algorithms for regression models. *IEEE Transactions on Fuzzy Systems*, 23:2343 – 2357, 2015.
4. J. M. Dennis, J. Edwards, K. J. Evans, O. Guba, P. H. Lauritzen, A. A. Mirin, A. St-Cyr, M. A. Taylor, and P. H. Worley. CAM-SE: A scalable spectral element dynamical core for the community atmosphere model. *Intl. Journal of High Performance Computing Applications*, 26:74–89, 2012.
5. G. E. Fagg, J. Pjesivac-grbovic, G. Bosilca, J. J. Dongarra, and E. Jeannot. Flexible collective communication tuning architecture applied to OpenMPI. In *In 2006 Euro PVM/MPI*, 2006.
6. B. Jackson, J. D. Sargle, D. Barnes, S. Arabhi, A. Alt, P. Gioumousis, E. Gwin, P. Sangtrakulcharoen, L. Tan, and T. T. Tsai. An algorithm for optimal partitioning of data on an interval. *Signal Processing Letters*, 12(2):105–108, 2005.
7. R. Killick, P. Fearnhead, and I. Eckley. Optimal detection of change points with a linear computational cost. *Journal of the American Statistical Association*, 107:1590–1598, 2012.
8. A. Scott and M. Knott. A cluster analysis method for grouping means in the analysis of variance. *Biometrics*, 30:507–512, 1974.
9. H. Steve. *Intel*® MPI library collective optimization on the Intel Xeon Phi coprocessor using environment variable collective operation control, 2015.
10. R. Thakur and W. Gropp. Improving the performance of collective operations in MPICH. *Proceedings of the 10th European PVM/MPI Users Group Meeting (Euro PVM/MPI 2003)*, pages 257–267, 2003.
11. B. Zhang, J. Geng, and L. Lai. Change-point estimation in high dimensional linear regression models via sparse group LASSO. *53rd Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 815–821, 2015.