Performance-influencing Factors for Parallel and Algorithmic Problems in Multicore Environments

Work-In-Progress Paper

Markus Frank & Steffen Becker
Universität Stuttgart
Stuttgart, Germany
(firstname).(lastname)@iste.uni-stuttgart.de

Angelika Kaplan & Anne Koziolek
Karlsruhe Institute of Technology
Karlsruhe, Germany
(firstname).(lastname)@kit.edu

ABSTRACT

Model-based approaches in Software Performance Engineering (SPE) are used in early design phases to evaluate performance. Most current model-based prediction approaches work quite well for single-core CPUs but are not suitable or precise enough for multicore environments. This is because they only consider a single metric (i.e., the CPU speed) as a factor affecting performance. Therefore, we investigate parallel-performance-influencing factors (PPIFs) as a preparing step to improve current performance prediction models by providing references curves for the speedup behaviour of different resource demands and scenarios. In this paper, we show initial results and their relevance for future work.

CCS CONCEPTS

• **Software and its engineering** → **Software performance**; • **Theory of computation** → *Parallel algorithms.*

KEYWORDS

Software Performance Engineering, Multicore, Performance-influencing Factors, Performance Prediction, Parallel Algorithms

ACM Reference Format:
Markus Frank & Steffen Becker and Angelika Kaplan & Anne Koziolek. 2019. Performance-influencing Factors for Parallel and Algorithmic Problems in Multicore Environments: Work-In-Progress Paper. In *Tenth ACM/SPEC International Conference on Performance Engineering Companion (ICPE '19 Companion), April 7–11, 2019, Mumbai, India.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3302541.3313099

1 INTRODUCTION

To handle the rising complexity and the demand for high-quality software, multiple disciplines are involved within a software development process. The one we are focusing in this paper is Software Performance Engineering (SPE). In SPE, model-based approaches are used to evaluate the quality attributes (i.e., scalability, elasticity, or performance) of software systems already during design time.

In the past, various approaches were developed to support software performance engineers and to enable accurate predictions even for complex, cloud-based systems. The most recent and most sophisticated approach are CloudSim [6] and Palladio [3]. These approaches use software-, hardware-, and user-models to model a given use-case and use analytic or simulation-based solvers to evaluate the software behaviour, quality attributes, and hardware utilisation. All these approaches use the CPU-speed as a single metric for performance, which works well for single-core-based environments. However, the performance of multicore-based environments is influenced by a plurality of influencing factors. Therefore, these approaches are not suitable for multicore-based environments because of their inaccurate performance predictions [4, 5].

As an overall goal, we aim for an improved multi-metric performance prediction model, which consists of some most relevant performance-influencing factors. Further, we aim to provide reference speedup curves for different kinds of scenarios and resource demands to further improve performance predictions.

In this paper, we investigate **P**arallel-**P**erformance-**I**nfluencing **F**actors (PPIFs) and their impact on the software performance, and how they can be described in performance prediction models. We provide a research strategy to collect PPIFs and an experimental setup to measure their impact. Further, we present and discuss preliminary measurements, results, conclusions, and sketch out the next steps in remaining work.

The paper is structured as follows: First, we give a brief overview of related work (Sec. 2), before we give our research approach and experimental setup in detail (Sec. 3). Next, we provide a first insight into our results and discuss them briefly (Sec. 4). We continue with a sketch of remaining work (Sec. 5) and a short conclusion (Sec. 6).

2 RELATED WORK

There are three main areas related to our work.

First, the research objective of performance-influencing factors in general is one of them. Existing papers dealing with this objective can be found in various domains (e.g., high performance computing and cloud computing), depending on a particular parallelisation paradigm (cf. [14]) or with focus on hardware, physical properties (cf. [9]). In this paper, we intent to consider PPIFs in a more general and holistic manner regarding these factors on hardware and software level (cf. Section 4.1).

Second, in reference to software performance prediction models, we can regard machine modelling approaches in algorithm engineering (cf. [12]) as another related topic to our work. These models play a major role when it comes to the analysis and optimisation of parallel algorithms. Simple, but adequately accurate machine models for parallel computer architectures are required (cf. [1], [12]). However, these models do not aim at making timing predictions.

ICPE '19 Companion, April 7–11, 2019, Mumbai, India

ACM ISBN 978-1-4503-6286-3/19/04…$15.00
https://doi.org/10.1145/3302541.3313099

Third, CPU simulators are used to design new microarchitectures and to predict the performance of an application with relatively high accuracy [10]. These type of simulators are often used by computer-system architecture research. In contrast to model-based performance prediction, CPU simulators require implementation of software or at least an execution trace [11]. Therefore, CPU simulators are unsuitable for our scope. However, we will analyse CPU simulators to learn about PPIFs.

3 METHOD AND EXPERIMENTAL SETUP

In the following, we outline our research approach and explain our experimental setup.

3.1 Research Approach

To improve the current performance prediction models, we propose to follow the method for experiment-based performance model derivation given by Happe [7]. Following this method, the performance model is extended step by step by additional PPIFs (e.g., cache size, memory bandwidth, etc.) until the accuracy reaches a satisfying level. However, knowing which PPIFs have the most significant impact on performance and therefore need to be considered in performance predictions is not trivial. A lot of different performance-influencing factors exist, and based on given different use-cases, they have a variable impact.

Therefore, in a first step we collect a set of PPIFs as complete as possible. For that, we (a) perform a structured literature review and (b) interview experts from the SPE, HPC, and OS domain. In a second step, we prioritise the PPIFs based on the results from the interviews. In the third step, we setup an experiment environment, where we vary only one of the PPIFs while we keep the others constant and evaluate the performance of this run. In the last step, we evaluate the measurements and provide performance reference curves for different scenarios, resource demands, and configurations.

3.2 Experimental Setup

Figure 1 shows a sketch of our experimental setup. To get accurate and reproducible results, we use ProtoCom[1]. ProtoCom enables us to generate work packages of specific primitive resource demand (like calculating Fibonacci numbers or primes). The advantage of using ProtoCom is that we can specify the exact runtime (i.e., five seconds) of these packages in a given environment [2]. This allows us to minimise variance.

For each resource demand, we use ProtoCom to generate various work packages of the same demand. In a next step, we take these packages and put them in a parallel execution paradigm (i.e., Java Threads). Next, we vary the performance-influencing factors. To do so, we fix all PPIFs except one (i.e., the thread number) and let them execute multiple times on our test environment. During the test run, we execute each setting 30 times and vary the one PPIF we are interested in.

4 PRELIMINARY RESULTS

In the following, we present preliminary results of our current work. First, we summarise the outcome of the PPIF-collection and the interviews. After that, we present measurements we took from one experimental setup and run.

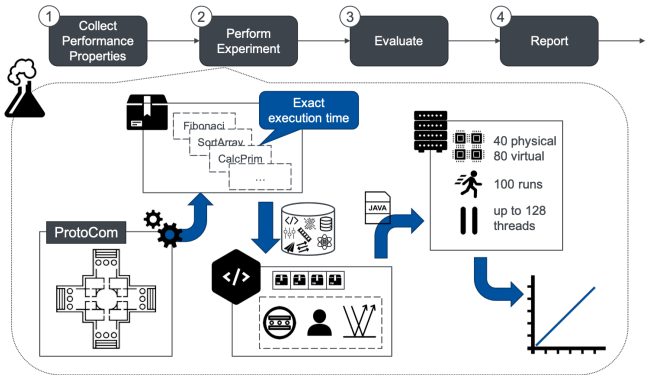

Figure 1: Sketch of the experimental setup

4.1 PPIF-Collection

The following list of PPIFs represents the outcome of a structured literature review and expert interviews we performed. For the latter, we interviewed four software performance experts within our department, seven HPC experts from the University of Dresden, Hasso-Plattner Institute, and Karlsruhe Institute of Technology (KIT), and three experts for parallel execution in embedded systems from the University of Chemnitz.

The following list is categorised into two groups and contains a subset of all PPIFs, which the experts agreed on.

4.1.1 Configurable PPIFs. Configurable factors are properties which can be directly configured or influenced by the software developer and therefore adjusted to the given hardware or scenario. Often auto-tuners are used to find the best configuration for these properties on a given system.

Parallelisation Strategy: The parallelisation strategy describes the used parallelisation paradigm or pattern. For example, Java Threads with a master-worker pattern, OpenMP, or ACTORS.
Thread Pool Size: The thread pool size specifies the number of worker threads. Typically, software threads are mapped to worker threads and then to hardware threads. Only worker threads are active.
Number of Threads: This is the number of totals spawned threads in the application. I.e., in a Java application spawning a thread for each task executed in parallel is possible. By using a thread pool, these threads are scheduled.
Software Caches: Software caches can influence the performance of the software significantly.
Data Locality: Usually data is stored in the memory belonging to the core which first touches/creates the data. So, this core has the optimal latency to access the data. Other cores have significant latency.

4.1.2 Fixed PPIFs. In contrast to configurable PPIFs, fixed PPIFs are given by the considered application or the used infrastructure and cannot be influenced by the software developer.

[1] https://sdqweb.ipd.kit.edu/wiki/ProtoCom

Type of Resource Demand: The type of resource demand is given by the type of the task performed on the CPU. I.e., processor-intensive tasks (like calculating Fibonacci numbers) or I/O-intensive tasks (like sorting an array).
Memory Design: The memory design is a hardware specific characteristic and defines the layout of CPUs, caches, and main memory. It also defines how these components are interconnected.
Memory Bandwidth: The memory bandwidth specifies the characteristics of the interconnections of the memory design, i.e., how many lanes are available, what is the total throughput, and how many components share the connection.

With these list of PPIFs at hand, we execute our experimental setup to determine the impact of each factor on the overall performance of the software.

4.2 Performance Measurements

In this section, we describe and report about our first experiments and findings. According to our experimental setup, we picked a first set of PPIFs—the number of worker threads and the type of resource demand—to focus on and executed a first experiment run.

Soft- and Hardware Setup. In our experimental setup, we executed a number of experiment runs. In each experiment run, we changed the resource demands and for each demand the number of worker threads. As resource demands, we used calculating Mandelbrot, Fibonacci, and prime numbers as well as sorting the array and counting numbers. Regarding the number of worker threads, we increased the configuration step-wise from 1 to 128 and as parallelisation paradigm we used Java Threads from the Java concurrent base class. Further, we configured ProtoCom do generate 1280 work packages each have a calibrated single thread execution time of 200 ms. In each run, we measured the completion time, i.e. the time needed to execute all work packages.

We executed all experiments on a dedicated server, which had 40 physical cores and hyper-threading enabled: 4x Intel(R) Xeon(R) CPU E7 - 4870 @ 2.40 GHz with 10 Cores each; Each CPU had an L1 Cache with 320 KiB, an L2 Cache with 2560 KiB, and an L3 Cache with 30 MiB; A total of 896 GB DDR3 RAM; Running on Ubuntu 16.04.

Results. The result of our measurements is plotted in Figure 2. The figure shows the speedup curves for the individual resource demands by using a different number of worker threads. Each data point represents speedup in comparison to sequential execution. To minimise variances, we used the mean execution time from the 30 experiment runs for each configuration to calculate the speedup.

The chart shows a near to linear speedup until 36 worker threads. As reference to calculate the speedup we used the execution time from a sequential execution. Staring from 40 worker threads, the speedup starts to spread a lot. Calculating Mandelbrot and sorting array demands continue to speed up while calculating primes and Fibonacci numbers do not benefit any more from additional worker threads. The counting numbers demand even drops significantly in performance. At around 80 worker threads we cannot observe an additional speedup.

Interpretation. In the following, we briefly interpret the observations.

Linear Speedup: The close-to-linear speedup of all demands up to 36 cores was expected since we used independent work packages, which do not interact with each other. Since we are using 40 physical cores, we expected all demands to speed up linear to this number of threads. However, the question remains why some demands start to slow down already before reaching 40 worker threads.
Hyper-threading: From 40 to 80 worker threads we can observe the additional benefit gained from hyper-threading. Especially the Mandelbrot and sorting array demand benefit from hyper-threading. Because these demands are an I/O-intensive task they benefit very well from hyper-threading.
Processor-intensive Tasks: While the Mandelbrot and sorting array demands benefit a lot from hyper-threading, calculating primes and Fibonacci numbers do not. The reason for this we see in the different characteristics of the demands. Mandelbrot and sort array are I/O-intensive demands. Calculating primes and Fibonacci numbers are processor-intensive tasks, which do not have to wait for I/O and therefore cannot take advantage of hyper-threading.
Performance Drop: The counting numbers demand is implemented to count from 0 to n while adding each number to a total sum. The reason for the performance drop of this demand is an open question and needs to be researched in further experiments.
Additional Worker Threads: We assumed using a higher number of worker threads than the number of cores available will result in a performance drop due to scheduling and spawning overhead. However, we could find no evidence for this. One reason for this might be that the number of 128 worker threads is only about 3x the number of physical cores. Further increasing the number of worker threads might provide evidence for this hypothesis.

5 REMAINING WORK

Remaining work includes extending the experimental setup along the three dimensions: (i) test environment, (ii) parallelisation strategy, (iii) complexity of the parallel and algorithmic problem.

In the following, we describe these dimensions in detail:

(i) Concerning the test environment, we plan to use different hardware setups and operating systems to compare and validate our preliminary results. As previously stated, we ran our experiments on a dedicated server, which is equipped as outlined in Section 4.2. In the next step, we will also apply our research approach (cf. Section 3.1) on a system called *bwUniCluster*. The extended HPC-system consists of more than 860 SMP-nodes with 64-bit Xeon processors from Intel. A detailed configuration description can be found in [13].

(ii) As mentioned before in Section 3.2, we have focused so far on Java Threads as an approach to parallelisation. Moreover, we will consider different kind of parallelisation strategy classes (i.e., shared, distributed, and hybrid memory parallelisation like combining MPI and OpenMP) in future work.

(iii) The last dimension refers to the complexity of parallel and algorithmic problems. Therefore, we distinguish between low, medium, and high complexity. So far, we considered low complexity problems (i.e., Mandelbrot, sorting array, calculating primes, Fibonacci numbers, and counting numbers; cf. Section 4.2) in our experiments. Medium complexity problems includes algorithms in terms of standard or optimised matrix-matrix multiplication (e.g.,

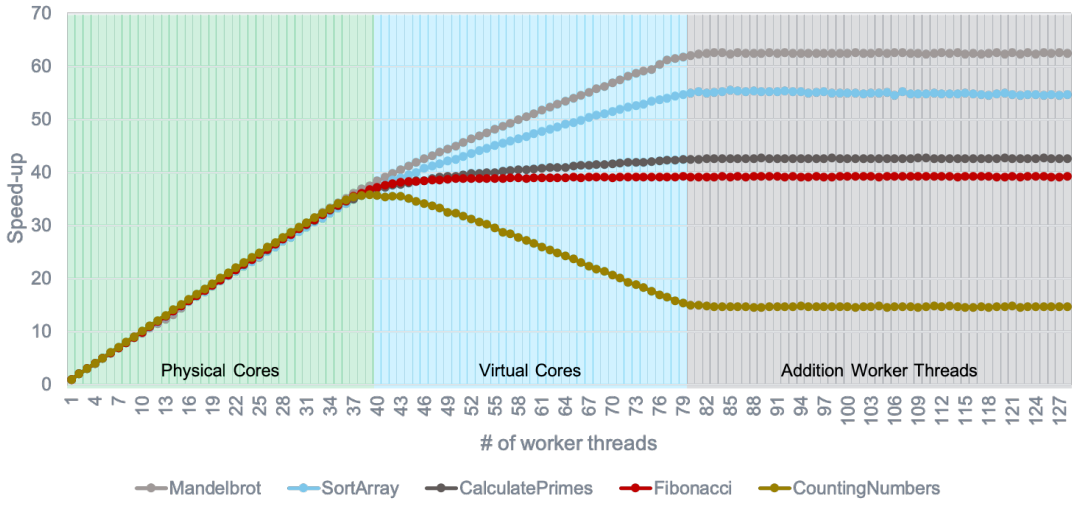

Figure 2: Measurements of Speedup Functions for Different Resource Demands on a 40-Core System with Enabled Hyper-threading

Strassen's matrix multiplication) and matrix inversion. As parallel matrix operations can be regarded as one of the most fundamental problems studied in scientific computing and are of great practical interest, they get more attention in future work.

Related to high complexity, we plan to investigate typical parallel algorithms in material science simulation. Therefore, we consider *Pace3D* (Parallel algorithms for crystal evolution in 3D) that contains different modules for the solution of diverse applications [8]. We will focus on the solver for phase-field models for microstructure formations in multi-component and multi-phase materials [8].

Furthermore, we extend our investigations along these dimensions regarding the remaining configurable PPIFs (i.e., software caches und data locality).

6 CONCLUSION

To improve the accuracy of current software performance models, we focused on parallel-performance-influencing factors (PPIFs). As a preparing step, we collected a set of PPIFs using a structured literature review and expert interviews. Dividing these factors in configurable and fixed, we analysed our PPIFs in-depth by defining an experimental setup using ProtoCom to get precise and reproducible results. In general, our setup is defined along three dimensions: (i) test environment, (ii) parallelisation strategy, (iii) complexity of the parallel and algorithmic problem. So far, we executed all experiments on a dedicated server regarding low complexity problems using Java Threads as parallelisation strategy. Thus, we provided reference curves for the speedup behaviour in our preliminary results. Finally, we gave an overview of remaining work to achieve our overall goal for an improved multi-metric performance prediction model, which consists of a number of most relevant performance properties.

REFERENCES

[1] David A Bader, Bernard ME Moret, and Peter Sanders. 2002. Algorithm engineering for parallel computation. In *Experimental Algorithmics*. Springer, 1–23.

[2] Steffen Becker, Tobias Dencker, and Jens Happe. 2008. Model-Driven Generation of Performance Prototypes. In *Proceedings of the SPEC International Workshop on Performance Evaluation: Metrics, Models and Benchmarks (SIPEW '08)*. Springer-Verlag, Berlin, Heidelberg, 79–98. https://doi.org/10.1007/978-3-540-69814-2_7

[3] Steffen Becker, Heiko Koziolek, and Ralf Reussner. 2009. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software* 82, 1 (2009), 3–22.

[4] Markus Frank and Marcus Hilbrich. 2016. Performance Prediction for Multicore Environments—An Experiment Report. In *Proceedings of the Symposium on Software Performance 2016, 7-9 November 2016, Kiel, Germany.* https://sdqweb.ipd.kit.edu/typo3/sdq/fileadmin/user_upload/palladio-conference/2016/papers/Performance_Prediction_for_Multicore_Environments_-_An_Experiment_Report.pdf

[5] Markus Frank, Stefan Staude, and Marcus Hilbrich. 2017. Is the PCM Ready for ACTORs and Multicore CPUs? — A Use Case-based Evaluation. In *Proceedings of the Symposium on Software Performance 2017, 9-10 November 2017, Karlsruhe, Germany.*

[6] Tarun Goyal, Ajit Singh, and Aakankasha Agrawal. 2012. Cloudsim: simulator for cloud computing infrastructure and modeling. *Procedia Engineering* 38, 4 (2012), 3566–3572.

[7] Jens Happe. 2008. *Predicting Software Performance in Symmetric Multi-core and Multiprocessor Environments.* Dissertation. University of Oldenburg, Germany. http://oops.uni-oldenburg.de/827/1/happre08.pdf

[8] Johannes Hötzer, A Reiter, H Hierl, Philipp Steinmetz, Michael Selzer, and Britta Nestler. 2018. The parallel multi-physics phase-field framework Pace3D. *Journal of computational science* 26 (2018), 1–12.

[9] Zheng Li, Liam OBrien, Rainbow Cai, and He Zhang. 2012. Towards a taxonomy of performance evaluation of commercial Cloud services. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*. IEEE, 344–351.

[10] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. 2005. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *SIGARCH Comput. Archit. News* 33, 4 (Nov. 2005), 92–99. https://doi.org/10.1145/1105734.1105747

[11] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-core Systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 475–486. https://doi.org/10.1145/2485922.2485963

[12] Peter Sanders. 2010. Parallel Algorithm Engineering. (2010). https://pdfs.semanticscholar.org/d98d/ff63386f1e2f629f1f480f7eb5d1119fdf3d.pdf

[13] SCC. 2018. KIT - SCC - Dienste - Wissenschaftliches Rechnen - High Performance Computing (HPC) und Clustercomputing - bwUniCluster. https://www.scc.kit.edu/dienste/bwUniCluster.php.

[14] Felix Wolf and Bernd Mohr. 2003. Automatic performance analysis of hybrid MPI/OpenMP applications. *Journal of Systems Architecture* 49, 10-11 (2003), 421–439.