

Sandra Wienke

Productivity and Software Development Effort Estimation in High-Performance Computing

Productivity and Software Development Effort Estimation in High-Performance Computing

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der
RWTH Aachen University zur Erlangung des akademischen Grades
einer Doktorin der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Sandra Juliane Wienke, Master of Science

aus Berlin-Wedding

Berichter: Universitätsprofessor Dr. Matthias S. Müller
Universitätsprofessor Dr. Thomas Ludwig

Tag der mündlichen Prüfung: 18. September 2017

Diese Dissertation ist auf den Internetseiten der Universitätsbibliothek online
verfügbar.

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.ddb.de> abrufbar.

Sandra Wienke:

Productivity and Software Development Effort Estimation in High-Performance Computing

1. Auflage, 2017

Gedruckt auf holz- und säurefreiem Papier, 100% chlorfrei gebleicht.

Apprimus Verlag, Aachen, 2017

Wissenschaftsverlag des Instituts für Industriekommunikation und Fachmedien
an der RWTH Aachen

Steinbachstr. 25, 52074 Aachen

Internet: www.apprimus-verlag.de, E-Mail: info@apprimus-verlag.de

Printed in Germany

ISBN 978-3-86359-572-2

D 82 (Diss. RWTH Aachen University, 2017)

Abstract

Ever increasing demands for computational power are concomitant with rising electrical power needs and complexity in hardware and software designs. According increasing expenses for hardware, electrical power and programming tighten the rein on available budgets. Hence, an informed decision making on how to invest available budgets is more important than ever. Especially for procurements, a quantitative metric is needed to predict the cost effectiveness of an HPC center.

In this work, I set up models and methodologies to support the HPC procurement process of German HPC centers. I model cost effectiveness as a productivity figure of merit of HPC centers by defining a ratio of scientific outcome generated over the lifetime of the HPC system to its total costs of ownership (TCO). I further define scientific outcome as number of scientific-application runs to embrace the multi-job nature of an HPC system in a meaningful way. I investigate the predictability of the productivity model's parameters and show their robustness towards errors in various real-world HPC setups. Case studies further verify the model's applicability, e.g., to compare hardware setups or optimize system lifetime.

I continue to investigate total ownership costs of HPC centers as part of the productivity metric. I model TCO by splitting expenses into one-time and annual costs, node-based and node-type-based costs, as well as, system-dependent and application-dependent costs. Furthermore, I discuss quantification and predictability capabilities of all TCO components.

I tackle the challenge of estimating HPC software development effort as TCO component with increasing importance. For that, I establish a methodology that is based on a so-called performance life-cycle describing the relationship of effort to performance achieved by spending the respective effort. To identify further impact factors on application development effort, I apply ranking surveys that reveal priorities for quantifying effects. Such an effect is the developer's pre-knowledge in HPC whose quantification is addressed by confidence ratings in so-called knowledge surveys. I also examine the quantification of impacts of the parallel programming model by proposing a pattern-based approach. Since meaningful quantifications rely on sufficient and appropriate data sets, I broaden previous human-subject based data collections by introducing tools and methods for a community effort.

Finally, I present the applicability of my methodologies and models in a case study that covers a real-world application from aeroacoustics simulation.

Zusammenfassung

Mit dem stetig wachsenden Bedarf an Rechenleistung erhöhen sich die elektrische Leistung und die Hardware- und Softwarekomplexität im Hochleistungsrechnen (HPC). Die steigenden Kosten für Maschinerie, Energie und Programmierung belasten dabei verfügbare Budgets zunehmend. Infolgedessen werden auch fundierte Entscheidungskriterien für die Verwendung vorhandener Budgets bedeutsamer. Besonders in Beschaffungsprozessen werden quantitative Metriken benötigt um die Kostenwirksamkeit eines HPC Zentrums abzuschätzen.

In dieser Arbeit stelle ich Modelle und Methodiken zur Unterstützung von Beschaffungen in deutschen, universitären HPC Zentren auf. Ich modelliere ihre Kostenwirksamkeit als Produktivitätsmetrik mit dem Verhältnis von wissenschaftlichen Resultaten (erzeugt über die Lebenszeit des HPC Systems) zu seinen Gesamtkosten (TCO). Meine weiterführende Darstellung der wissenschaftlichen Resultate als aggregierte Ausführungszahl aller (Simulations-)Anwendungen auf dem HPC System deckt auch die typische Programm-Mischung eines HPC Clusters ab. Ich untersuche die Vorhersagbarkeit der Parameter des Produktivitätsmodells und zeige, dass das Modell robust gegenüber Fehlern in realen HPC Setups ist.

Als Teil der Produktivitätsmetrik führe ich ein Gesamtkosten-Modell eines HPC Zentrums ein, das zwischen einmaligen und jährlichen Kosten, zwischen Ausgaben basierend auf der Rechenknotenzahl und dem Rechenknotentyp, sowie zwischen system- und anwendungsabhängigen Kosten differenziert. Darüber hinaus diskutiere ich mögliche Quantifizierungen und Vorhersagbarkeiten der TCO-Komponenten.

Da die Wichtigkeit des Entwicklungsaufwands als Gesamtkostenteil steigt, führe ich eine Methodik zur HPC Aufwandsschätzung basierend auf einem sog. Performance-Lebenszyklus ein, der den Zusammenhang zwischen Entwicklungszeit für die wissenschaftliche Anwendung und erreichter Maschinen-Leistung beschreibt. Zur Identifizierung von Faktoren mit Einfluss auf den Aufwand dienen Umfragen zur Prioritäten-Bildung. Beispielsweise ist das HPC Vorwissen eines Entwicklers eine Schlüsselkomponente, die ich mit Umfragen zur Wissens-Selbsteinschätzung angehe. Ebenfalls untersuche ich den Einfluss des parallelen Programmiermodells. Da aussagekräftige Quantifizierungen auf ausreichenden Daten beruhen, motiviere ich ihre Erhebung in einem HPC-weiten Ansatz durch die Unterstützung von Werkzeugen und Methoden.

Schließlich stelle ich eine Fallstudie aus dem Bereich der Aeroakustik-Simulation vor, in der ich die Anwendbarkeit meiner Methodiken und Modelle darstelle.

Acknowledgements

This work would have never been successful without the incredible support and encouragement of my significant other, family, friends, colleagues, and supervisors.

I would like to thank the IT Center (formerly known as the Center for Computing and Communication), the Chair for High Performance Computing and the Aachen Institute for Advanced Study in Computational Engineering Science (AICES) of RWTH Aachen University for the financial support during my years of research. This includes the opportunities for traveling as ground base for getting food for thought from different perspectives. In addition, I really appreciate the fruitful discussions, feedback and mentoring from my supervisors Prof. Matthias S. Müller and Prof. Thomas Ludwig, as well as the inspiration provided by Prof. Christian Bischof and Dieter an Mey who initially pointed me to the field of productivity and TCO evaluations in HPC. I particularly thank Christian Terboven for his guidance in scientific and everyday life, as well as his effort in proof-reading my thesis. My thanks also go to Dirk Schmidl for his criticism and critique that improved my work and for our verbal exchanges that spiced up the workday. Further, I would like to acknowledge Julian Miller who contributed to the methodological approach of effort estimation—even through overseas task arrangements. Additionally, Tim Cramer helped to set up and supervise software labs starting in 2013 that was a ground base of my effort evaluations: Thanks. I thank Paul Kapinos for providing supplies for the ‘Snäckbär’ and the administrators of the IT Center for providing short-notice trouble shooting if needed. To all my colleagues who made an outstanding good working atmosphere possible: Thank you—Dieter an Mey, Tim Cramer, Alesja Dammer, Hristo Iliev, Jannis Klinkenberg, Julian Miller, Joachim Protze, Dirk Schmidl, Daniel Schürhoff, Aamer Shah, Christian Terboven and Bo Wang. Moreover, I am grateful to our MATSE apprentices and student workers for their supportive work, as well as to my former colleague Oliver Fortmeier for the great Latex thesis template.

My internship at the Lawrence Livermore National Laboratory, USA, enabled me to drive my research on software development effort estimation and programmability forward, especially due to the support of Prof. Martin Schulz. Thank you, for giving me the opportunity to focus on this topic, for connecting me to different researchers and for improving my scientific writing skills. I also appreciate the various GPU hackathon organizers who have given me the chance to

integrate my research on human-subject based effort quantification into the corresponding events. Prof. Sunita Chandrasekaran has been a very strong supporter of my person which increased my confidence and especially fostered my scientific skills in reviewing and chairing. It has always been fun working and spending time with you. Thanks a lot! I further would like to express my thankfulness to Michael Klemm for his (sometimes hard-drinking) customer service, to Prof. Gerhard Dikta and Benjamin Weyers who provided me with insights into statistical analysis, and to all other people who have been part of my research such as (domain) developers, students and survey takers.

My appreciation is also expressed to all my friends for their understanding and support. Special thanks go to Nicole, Sven, Laura and Lars for the varied conversations and (childlike) lightheartedness that provided a lot of fun. I also thank my handball team and aerobics allies who helped me to clear my mind after long working sessions.

Last but not least, I express my deep gratitude to the most important people in my life who supported me in any situation and encouraged me to go my way. My parents Irene and Wolfgang have aroused my curiosity to try things out and excited me about maths, programming and computer science early on in my childhood. During my doctoral studies, they provided mental support, invested effort to lighten my organizational workload or took care of my well-being by providing care packages. Thank you so much, mum and dad! I also thank my grandmothers Rita and Elfriede who always believed in me! Finally, I cannot thank enough my significant other! He had great patience and understanding on long working days. He affirmed me, encouraged me and cheered me up. I love you, Benno. I would have never been able to keep up with this without you.

Contents

List of Figures	xiii
List of Tables	xv
1. Introduction	1
1.1. Motivation	1
1.2. Contributions	4
1.3. Structure	5
 A. Productivity of HPC Centers	 9
2. Motivation	11
2.1. Definition of Productivity in Economics	11
2.2. Cluster Procurement in Germany	12
2.3. Developer's Effort Optimization	14
3. Related Work	17
3.1. The HPCS Program	18
3.2. User Perspective	18
3.2.1. Relative Development Time Productivity	18
3.2.2. Relative Power & Relative Efficiency	20
3.2.3. Response Time Oriented Productivity	21
3.3. System Perspective	22
3.3.1. Utility Theory	22
3.3.2. Throughput Oriented Productivity	23
3.3.3. SK ³ Synthesis Productivity	24
3.4. Business Perspective	25
3.4.1. Benefit-Cost Ratio	26
3.4.2. Return on Investment	26
3.5. Other Approaches	27
3.5.1. Benchmark Suites	28
3.5.2. Research Competitiveness	29
3.5.3. Cloud Computing	30

4. Modeling Productivity in HPC	31
4.1. Definition of Productivity	32
4.2. Value: Number of Application Runs	34
4.2.1. Single-Application Perspective	34
4.2.2. Composition to Job Mix	36
4.3. Cost: Total Cost of Ownership	39
4.3.1. Single-Application Perspective	40
4.3.2. Composition to Job Mix	43
4.4. Discussion of Parameters	43
4.4.1. Number of Nodes	44
4.4.2. System Lifetime	45
4.4.3. Number of Application Runs	46
4.4.4. Total Ownership Costs	49
4.5. Derived Metrics	50
4.6. Uncertainty and Sensitivity Analysis	52
4.6.1. Uncertainty	52
4.6.2. Sensitivity	54
4.6.3. Results	58
4.7. Tool Support	66
 B. Total Cost of Ownership of HPC Centers	 69
5. Motivation	71
6. Related Work	75
6.1. Data Centers	76
6.2. HPC Centers	78
7. Modeling Total Cost of Ownership in HPC	79
7.1. Definition of TCO	79
7.1.1. Single-Application Perspective	79
7.1.2. Composition to Job Mix	81
7.2. Discussion & Quantification of Components	82
7.2.1. Hardware Purchase Costs	82
7.2.2. Hardware Maintenance Costs	82
7.2.3. Software and Compiler Costs	83
7.2.4. Infrastructure Costs	84
7.2.5. Environment Installation Costs	85
7.2.6. Environment Maintenance Costs	86
7.2.7. Development Costs	86
7.2.8. Application Maintenance Costs	87
7.2.9. Energy Costs	87
7.2.10. Other Parameters	89

C. Development Effort Estimation in HPC	91
8. Motivation	93
8.1. Definition of Development Effort in HPC	93
8.1.1. Development Time	94
8.1.2. Time to Solution	96
8.2. Estimation of Development Time	97
8.2.1. Current Techniques in HPC	97
8.2.2. Performance Life-Cycle	97
9. Related Work	99
9.1. Software Complexity Metrics	99
9.1.1. Lines of Code	99
9.1.2. Function Points	102
9.1.3. Halstead Complexity Metric	104
9.1.4. Cyclomatic Complexity Metric	106
9.2. Software Cost Estimation Techniques	108
9.2.1. COCOMO II	108
9.2.2. Uncertainty and Accuracy	114
10. Methodology of Development Effort Estimation in HPC	117
10.1. Methodology Step by Step	118
10.2. Effort-Performance Relationship	120
10.2.1. Performance Life-Cycle	120
10.2.2. Related Work	121
10.2.3. Statistical Methods	122
10.2.4. Proof of Concept	122
10.3. Identification of Impact Factors	124
10.3.1. Ranking by Surveys	124
10.3.2. Related Work	125
10.3.3. Statistical Methods	126
10.3.4. Proof of Concept	127
10.4. Quantification of Impact Factor “Pre-Knowledge”	128
10.4.1. Knowledge Surveys	129
10.4.2. Related Work	130
10.4.3. Statistical Methods	132
10.4.4. Proof of Concept	132
10.5. Quantification of Impact Factor “Parallel Programming Model & Algorithm”	134
10.5.1. Pattern-Based Approach	134
10.5.2. Related Work	135
10.5.3. Proof of Concept	137

11.Data Set Collection	139
11.1. Related Work	139
11.1.1. Manual Diaries	139
11.1.2. Automatic Tracking	140
11.2. EffortLog—An Electronic Developer Diary	142
11.2.1. Tool Description	142
11.2.2. Proof of Concept	143
11.3. Challenges	145
11.3.1. Trust in Data	145
11.3.2. Data Gathering	146
 D. Making the Business Case	 149
12.Aeroacoustics Simulation Application—ZFS	151
12.1. Description	151
12.2. HPC Activities	152
13.Development Effort	153
13.1. Knowledge Survey	153
13.2. Performance Life-Cycle	153
14.Total Costs	157
14.1. System-Dependent Components	157
14.2. Application-Dependent Components	157
15.Productivity	159
15.1. Ex-Post Analysis	159
15.2. Ex-Ante Analysis	161
15.3. Future Directions	163
 16.Conclusion	 165
 Bibliography	 169
 Acronyms	 193
 Appendix	 197
A.1. Use Cases	197
A.1.1. Neuromagnetic Inverse Problem—NINA	197
A.1.2. Engineering Application—psOpen	199
A.1.3. Conjugate Gradient Solver	201
A.1.4. Hardware Setup	204

A.2. Productivity Evaluation	205
A.2.1. Parameters & Predictability	205
A.2.2. Parameters for RWTH Aachen University Setups	206
A.2.3. Tools	209
A.3. Effort Estimation	211
A.3.1. COCOMO II Cost Drivers	211
A.3.2. Ranking of Impact Factors	213
A.3.3. Impact Factor “Pre-Knowledge”	214
A.4. Data Collection Tool	214
A.4.1. Characteristics of EffortLog	214
A.4.2. User Interface of EffortLog	216
A.5. Sensitivity Analysis by Saltelli	217
Statement of Originality	221

List of Figures

1.1. Top500 performance development over time	2
2.1. Productivity index of two plants over time	12
2.2. Exemplary productivity use cases in HPC	14
4.1. Projects at RWTH Compute Cluster ordered by core-h used	37
4.2. Cluster capacity and simplified sample share of applications	38
4.3. Comparison of productivity as function of investment with number nodes computed as integer values and real values	44
4.4. Schematic relationship of uncertainty based on PDFs	53
4.5. Uncertainty and sensitivity analysis of <i>psOpen</i> for the small-scale setup	62
4.6. PDFs of <i>NINA</i> with different HPC setups	63
4.7. CDFs of <i>NINA</i> with different HPC setups	63
4.8. Main and total effects of the full job mix	65
4.9. Productivity PDF of the full and reduced job mix	65
4.10. Productivity variation for 30,000 samples ordered by size: full and reduced job mix	65
4.11. PDF of the relative error between full and reduced job mix	65
5.1. Sample TCO shares for one year according to Jones and Owen	72
5.2. Sample TCO shares for one year according to Bischof et al.	72
8.1. Exemplary performance life-cycle with different milestones	98
9.1. Control flow graph for cyclomatic complexity of Pi example	107
9.2. Cone of Uncertainty	115
10.1. Different performance life-cycle approaches	120
10.2. Effort-performance relationships collected from student software labs with different parallel programming models	123
10.3. Ranking of impact factors based on 44 data sets	128
10.4. Means of pre-KS and post-KS results per knowledge question group based on 6 data sets	133
10.5. Variability of pre-KS means per knowledge question group across 7 student data sets	133

10.6. Programming models: p -values of one-sided Wilcoxon rank sum test with respect to students' development effort and runtime	137
12.1. Acoustic pressure field of a jet simulated with <i>ZFS</i>	152
13.1. Means of pre-KS and post-KS results per knowledge question group based on developer conducting <i>ZFS</i> GPU tuning	154
13.2. Performance life-cycle of <i>ZFS</i> with respect to GPU development activities	155
14.1. Trace of power consumption of original MPI version and tuned OpenACC version of <i>ZFS</i>	158
15.1. Productivity of <i>ZFS</i> setups	160
15.2. Estimation of productivity of <i>ZFS</i> setups	162
A.1. Sample view of the TCO spreadsheet with focus on a manager per- spective	209
A.2. Webpage view of the <i>Aachen HPC Productivity Calculator</i>	210
A.3. Ranking of impact factors for HPC professionals and students	213
A.4. Means of pre-KS and post-KS results per knowledge question group for HPC professionals and students	214
A.5. User interface of <i>EffortLog</i>	216

List of Tables

4.1. Possible TCO Components	40
4.2. Assumptions for PDFs of productivity parameters	59
5.1. Tier-2 HPC funding in Mio. € and according to the funding instrument of research buildings	71
8.1. Definition of development time: includes and excludes	94
9.1. Unadjusted function point count of Pi example	104
9.2. Halstead’s operators and operands of Pi example	105
9.3. Halstead’s operators and operands of Pi example including OpenMP pragma	105
9.4. Scale Factors and Effort Multipliers used in COCOMO II	109
9.5. Exemplary Likert-type rating scales used in COCOMO II	110
10.1. Overview of previous studies on performance life-cycles	119
10.2. Possible milestones in a performance life-cycle	121
10.3. Selected factors impacting development effort	125
10.4. 3-part rating scale for knowledge surveys	129
11.1. Comparison of tracked effort data using <i>EffortLog</i> and manual diaries for one hackathon event and two years of software labs	144
13.1. Comparison of tracked effort data using <i>EffortLog</i> and manual diaries for porting and tuning <i>ZFS</i> with OpenACC for GPUs	154
15.1. Application-dependent productivity parameters of <i>ZFS</i>	159
15.2. Performance estimations for <i>ZFS</i> based on Stream bandwidth measurements, and actual kernel runtime measurements	161
A.1. Application-dependent productivity parameters of <i>NINA</i> : effort, performance and power consumption for different HPC setups	199
A.2. Application-dependent productivity parameters of <i>psOpen</i> : effort, performance and power consumption for 2 implementation phases	201
A.3. Application-dependent productivity parameters of the <i>CG solver</i> : reference effort and performance for (hybrid) implementations with different parallel programming models	204

A.4. Hardware details for different architectures used in this work	204
A.5. Parameters of productivity model categorized by application-de- pendent and system-dependent components and their predictability capabilities	205
A.6. Summary of estimated or measured productivity parameters of both <i>psOpen</i> and <i>NINA</i>	208
A.7. Description of effort multipliers used in COCOMO II	211
A.8. Description of scale factors used in COCOMO II	212
A.9. Comparison of different approaches to collect effort data	215

1. Introduction

1.1. Motivation

Driven by complex scientific simulations, the ever increasing demands for computational power has led to an exponential growth in floating-point performance of HPC systems over the last decades. The Top500 list [TOP17] recognizes this trend since 1993 with a yearly performance growth rate of roughly 85 %. However, this performance growth is slowing down: Performance curves of the average and #500 systems show inflection points in 2013 and 2008 where trajectories reduce to 40 % and 54 %, respectively (compare Fig. 1.1). The Top500 co-founder Strohmaier attributes this falling trend also to power consumption and money, instead of sheer technological aspects [Tra15]. Indeed, in pursuit of next generation supercomputer deployments, vast investments for procuring and operating large-scale HPC systems are required. Especially, extrapolation of currently consumed power to the next generation would result in non-affordable sums of hundreds of million euros spent on energy per year. This heralded era with increased emphasis on budget constraints necessitates an informed decision making process in HPC procurements more than ever.

For procurements of large HPC systems being worth millions of euros, politics and advisory councils in Germany have a strong impact on fundings and external conditions. Since the Top500 list with Linpack performance numbers is a prominent ranking list pretending to reflect the worth of HPC machines, number-crunching machines have been emphasized for a long time. However, numerous scientific applications are today bound by memory bandwidth and suffer from inherent memory bottlenecks [AHL⁺11]. This discrepancy of Top500 machines and user requirements has also been recognized by funding agencies who increased the focus on supporting the production of scientifically valuable output.

Hence, emerging technological advances must keep a tight rein on power consumption and budgets while adhering to users' needs. The up-to-date nature of power, budget and memory bounds is exemplified by the rising interest of the HPC community in ARM's alternative low-power low-budget processor design: ARM has very recently extended its HPC ecosystem and will constitute a Cray supercomputer in future (summary given in [Rus17]). Likewise, bandwidth limits are

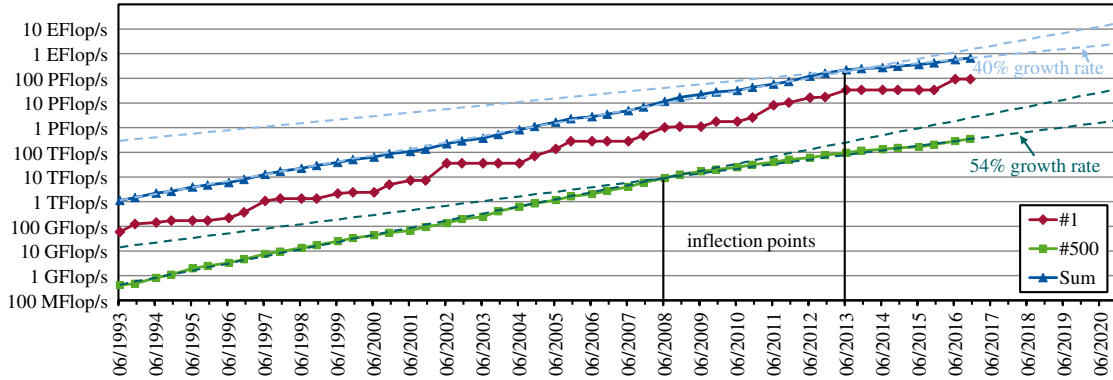


Figure 1.1.: Top500 performance development over time. Dashed lines approximate yearly growth rate similar to [SMDS15]. *Sum* systems represent $500\times$ the average.

addressed by current developments that focus on scratch pad, non-volatile or 3D stacked memories.

Having noted HPC budget constraints and required usefulness for maximized scientific outcome, HPC managers' decisions on how to invest available budgets need to be supported by an appropriate metric that covers these characteristics. The productivity figure of merit is widely used in economics for these purposes and defines the ratio of output units to input units, e.g., value (of scientific output) over cost. During the High Productivity Computing Systems (HPCS) program that focused on the development of economically viable HPC machines [DGH⁺08], different HPC productivity metrics have been proposed such as the (SK)³ model that captures relative power and efficiency of a programming interface in terms of useful work per time (e.g., Flop/s) over costs. However, with respect to available publications, existing models have only rarely been shown to be applicable in real-world HPC procurement setups. Possible shortcomings may be their complex structure, model parameter introduction without validation of their chances for quantification and predictability, neglect of job mix setups, or intangible metrics for evaluating the value of an HPC system.

Nevertheless, the cost component for productivity models of HPC centers, that provide HPC resources and services to researchers, remains undisputed. Moreover, with recognition of the increasing power expenses, sheer purchase-based cost calculations have been replaced by models of total costs of ownership (TCO) which incorporate capital one-time expenditures and operational annual costs. Since businesses are economically motivated to minimize their total costs, TCO models are also popular by any kind of software or hardware vendor to promote the benefits of their product in comparison to other solutions, though usually by limiting the TCO perspective to certain parts. In scientific setups, sound TCO evaluations comprise comprehensively all arising costs including expenses for manpower needed to enable scientific applications to efficiently exploit HPC systems. Un-

fortunately, this effort for parallelization, porting or tuning of applications has been taken for granted far too long and has only occurred as essential contributor to strategic developments in HPC in recent years. Bischof et al. [BaMI12] even showed in their TCO calculations that investments into HPC experts' brainware are recouped by cost savings through applications' performance gains.

The importance of embracing manpower efforts into well-founded TCO models even increases with the move to next generation supercomputer deployments. While meeting electrical power and budget constraints, HPC systems continue to increase in hardware and software complexity and, hence, development efforts spent for parallelizing, porting or tuning applications do. To avoid unproductive large-scale HPC systems, the U.S. Department of Energy (DOE) started engaging in the hardware/software co-design process [DOE17b], as well as the European Commission in their second Horizon 2020 work program [Eur17]. Additionally, recent recommendations for German HPC funding cover the inclusion of brainware costs to some extent [Wis15a].

Despite recognition of HPC software development efforts as crucial component for cost-effective HPC centers, HPC managers still face the challenge to estimate these efforts as foundation for informed procurement decisions. Although software cost estimation models are popular in mainstream software engineering (SE), they cannot be directly transferred to performance-critical HPC environments. In contrast to traditional software development, HPC activities usually build upon existing domain code that is further enhanced for efficient operation on parallel computer architectures, and focus on squeezing out the last percentage points of performance consuming much development time. Estimation challenges also arise from the numerous impact factors on HPC development effort. The amount of needed effort may be affected by the pre-knowledge of the developer, the kind of application, hardware architecture, parallel programming model, available tools landscape, maturity of compilers and so on. The majority of previous studies on HPC development effort investigated the influence of the parallel programming model while mostly ignoring possible variations in the other factors. Furthermore, only few effort studies have been conducted due to their overhead evoking nature. To rely estimation models on (statistically) meaningful data, broad collection of data sets must be fostered. If in place, such a model will not only be useful for procurement decisions, but also find use cases in industry setups for parallel software exploitation or in tool development and training environments to show the effectiveness of their product or educational program, respectively.

This work tackles the challenges of procurement decision making of German university HPC centers. It presents methodologies to evaluate their cost effectiveness by productivity and TCO models, and to estimate the effort of HPC-related activities. One focus is on their applicability to real-world setups illustrated by proof of concepts, practical case studies, and statistical analysis of collected data

sets. Spanning a broad field, the methodologies adapt successful concepts from economics, software engineering, and educational assessment, and cover strategies in human-subject research in HPC. Section 1.2 presents corresponding contributions of this work and, in Sect. 1.3, the thesis structure is described.

1.2. Contributions

The main contributions of this work are methodologies and models for productivity, total costs of ownership and development effort estimation in HPC as apparent in the three main parts of this work. My structure follows a hierarchical approach: Productivity — as defined by value over cost — represents a quantifiable metric to predict the worth of an HPC center depending on its utilizing mix of compute jobs (Part A). Its cost component can be modeled in terms of total ownership costs of HPC centers (Part B). For a TCO (and productivity) model with predictive power, concepts for estimation of all TCO components are required. This includes estimation of development efforts needed to parallelize, port and tune applications for HPC systems (Part C). Hence, the following contributions stand out:

Methodology for Productivity Evaluation in HPC Procurements In a methodological approach, I define a productivity model that enables the comparison of various HPC setups for procurement processes in German university HPC centers. To evaluate the value of an HPC center as part of the productivity model, I focus on scientific output and introduce a metric based on the amount of real-world application executions. This metric is easy to quantify and, especially, supports the investigation of multiple applications exploiting the HPC system. Thus, the productivity model presented in this work reflects the cost effectiveness of an HPC center that employs widely-differing job mixes. With emphasis on applicability in procurement processes, my methodology incorporates gathered experiences from RWTH Aachen University and suggests estimation strategies for all model parameters. An analysis of uncertainty and sensitivity of the productivity model illustrates the validity of my approach.

Model & Quantification of Total Ownership HPC Costs As part of the productivity model, total costs of ownership heavily contribute to the cost effectiveness of HPC centers. I present a TCO model for HPC centers that comes at a tradeoff between accuracy and over-parametrization, while still giving leeway for additional model parameter refinements. Besides splitting costs into one-time and annual expenses, I also interpret costs either as scaling factor with the number of compute nodes or as node-type dependent component. For all defined TCO components, I illustrate methods to quantify them by numerical values, based on data

collection at RWTH Aachen University. With respect to the required predictive power of the productivity model and, hence, of the TCO model, I further discuss estimation concepts for all TCO components.

Methodology for Development Effort Estimation in HPC Development effort needed to parallelize, port or tune applications for a particular HPC environment is defined as part of the TCO model and, thus, also affects procurement decisions. I provide a methodology to estimate HPC-related efforts that enables maintaining the predictive power of the productivity model. In this context, I evaluate methods from traditional software engineering in terms of software complexity metrics and software cost estimation for applicability to HPC projects. Because of their limited suitability, I establish the concept of *performance life-cycles* that represents the relationship of application performance and effort needed to achieve this performance. Due to its many-fold dependence on various impact factors, the identification of these effects is part of my methodology. For that, I introduce the design of ranking surveys and find developers' pre-knowledge and the applied parallel programming model to a certain numerical algorithm as two key drivers. The methodological concepts for quantification of these key drivers touch knowledge surveys and parallel patterns. Finally, I address the need to rely my methods on statistically meaningful data by presenting the tool *EffortLog* for tracking effort-performance pairs during HPC development, and by targeting at a community effort to collect sufficient data sets. Focusing on applicability in real-world HPC setups, I show proof of concepts and evaluate data sets obtained from case studies.

1.3. Structure

The following Part A focuses on productivity of HPC centers. I introduce the productivity metric as defined in economics in Sect. 2.1 and motivate its adaption for the HPC domain by illustrating its applicability in several use cases from HPC procurement (Sect. 2.2), but also from the user perspective (Sect. 2.3). Section 3 covers related work. Most other productivity models have been introduced in the context of the HPCS program (Sect. 3.1) and either with focus on the user perspective (Sect. 3.2), or from a system perspective (Sect. 3.3). Sections 3.4 and 3.5 present work outside of the HPCS program that investigates productivity as benefit-cost ratio or return on investment. My contributions to productivity modeling in HPC constitute Sect. 4. I start with the basic formulation of HPC productivity in Sect. 4.1, and provide further details of the model in Sect. 4.2 and 4.3 where I define value and costs from a single-application and multiple-application perspective. Section 4.4 elaborates on parameters of the productivity model and discusses their predictability. I present an uncertainty and sensitivity

analysis of these parameters in Sect. 4.6. Useful derivations of my productivity model are explained in Sect. 4.5, whereas exemplary implementations of the model in selected tools are described in Sect. 4.7.

Part B picks up the cost component of productivity. Increasing computational and electrical demands while ascertaining operation of HPC software on next generation hardware motivates the aggregation of all costs into one model (Sect. 5). While traditional data centers have some experiences in the application of TCO models (see Sect 6.1), research on TCO models for HPC centers is scarce as related work in Sect. 6.2 illustrates. Thus, my contributions describe a TCO model and quantification for HPC setups in Sect. 7. This includes the refinement of TCO parameters and relationships in Sect. 7.1 with respect to the productivity model's cost component. Section 7.2 presents suggestions for quantification and estimation of all TCO components.

In Part C, I investigate HPC software development effort as part of the TCO model. As reference for further evaluations, I define the meaning of HPC development effort in Sect. 8.1, and motivate estimation models based on effort-performance relationships (Sect. 8.2). Related work in Sect. 9 focuses on methods in mainstream SE and their applicability to HPC setups. In this context, I show shortcomings of traditional software complexity metrics, e.g., lines of code or function points, for HPC development in Sect. 9.1. I present the traditional software cost estimation model COCOMO II and its evaluation for HPC projects in Sect. 9.2. As major contribution, the methodology for development effort estimation in HPC is presented in Sect. 10. After an overview of all steps constituting the methodology (Sect. 10.1), I explain the relationship of effort and performance as base metric for my effort estimation model (Sect. 10.2). I express it by the invention of a *performance life-cycle* and illustrate it by results for human-subject case studies. Section 10.3 deals with the identification of impact factors on HPC development effort and shows early results based on ranking surveys. Having identified pre-knowledge as one key driver of effort, I borrow the concept of knowledge surveys from educational assessment and apply it for the quantification of pre-knowledge in HPC (Sect. 10.4). Another impact factor is given by the parallel programming model and the kind of numerical algorithm. For its corresponding quantification, I present ideas with focus on a pattern-based approach in Sect. 10.5. Since appropriate data sets are the groundwork for these evaluations, I cover strategies on data collection as part of my methodology in Sect. 11. To combine the advantages of manual and automatic data collection (Sect. 11.1), I introduce the electronic developer diary *EffortLog* in Sect. 11.2. Section 11.3 discusses challenges in the data collection process that mainly arise from work with human subjects.

In Part D, I make the business case by applying previously-introduced models and methodologies to a case study from aeroacoustics simulations in RWTH HPC setups. The application and respective HPC activities for leveraging accelerator-

type hardware are described in Sect. 12. Section 13 covers results from knowledge surveys and performance life-cycles for effort quantification. All remaining system-dependent and application-dependent TCO components for this case study are set up in Sect. 14, while Sect. 15 combines quantified values to productivity. Besides an ex-post procurement analysis (Sect. 15.1), it also evaluates the model's predictive power for next generation hardware in Sect. 15.2. Future directions for work with the simulation framework and for other productivity studies are given in Sect. 15.3.

Finally, I summarize my work in Sect. 16 and present future research.

Part A.

Productivity of HPC Centers

Focus on being productive instead of busy.

TIMOTHY FERRISS, AMERICAN
SCIENCE WRITER & AUTHOR OF
'THE 4-HOUR WORKWEEK'

2. Motivation

The metric *productivity* is prevailing in economics to measure the value of products and programs. Its economic definition (Sect. 2.1) illustrates challenges and lays the foundation of an application to HPC setups. Especially, with increasing expenses for hardware, energy and developer efforts, HPC decision makers have to argue how to invest available budgets. Naturally, they strive for investments that maximize the productivity of an HPC center. The most prominent use case and the focus of my work is the procurement of new cluster hardware that involves a comparison of different systems and setups (Sect. 2.2). Additionally, productivity can express the tradeoff between labor cost and performance that is achieved by investing effort for tuning HPC applications (Sect. 2.3).

2.1. Definition of Productivity in Economics

The intrinsic aim of decision makers in industry and for production processes in governments is to maximize their economic value and productivity. Despite its prevalence, the term productivity suffers from ambiguous definitions [Tan05]. The most basic definition of productivity in economics is defined as the ratio of units of output to units of input [Mod07]:

$$\text{productivity} = \frac{\text{outputs}}{\text{inputs}} \quad (2.1)$$

This productivity index is commonly reduced to labor or capital productivity, e.g., measured as output per labor hour or per machine hours used, respectively. A more comprehensive approach combines the contribution of several input factors to so-called *multifactor productivity* (*MFP*) (or also *total factor productivity*) [Che88][Mod07]. The denominator of the productivity index in Equ. (2.1) is then resolved into a single unit that is commonly expressed in dollar.

Closely-related to MFP is the term *cost effectiveness* (*CE*): The inputs (or costs) are clearly defined in monetary units, while the outputs (or benefits/ value) are measured in units other than money [CK10]:

$$\text{cost effectiveness ratio} = \frac{\text{value}}{\text{cost } [\text{\$}]} \quad (2.2)$$

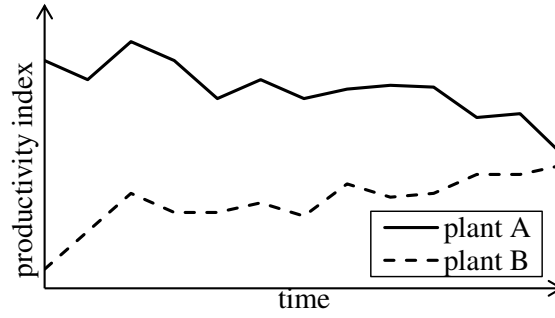


Figure 2.1.: Productivity index of two plants over time. Similar to [Che88].

I use the terms (multifactor) productivity and cost effectiveness interchangeably in the following. A diverging approach to cost-effectiveness analysis (CEA) is defined as cost-benefit analysis (CBA) and measures the benefits in Equ. (2.2) in monetary units, too. The corresponding benefit-cost ratio (BCR) degrades to a value without unit and represents an economic worth if $BCR > 1$ [CK10].

While statisticians come up with accurate MFP productivity models, e.g., in form of logarithmic or multiplicative production functions, the challenge is to introduce an index that decision makers are able to understand and relate to in order to affect their behavior as stated by Chew [Che88], a senior consultant and former associate professor at the Harvard Business School in Boston. Furthermore, the appropriate interpretation and usage of such an index is challenging [Che88][CK10]: First, a productivity index as single point does not reveal much without comparison to other firms or production setups. CEA even assumes that several competitive ways exist to fulfill the firm's objective. In contrast, BCR usually stands alone and states either worthiness or worthlessness of an approach without reflecting on any alternatives. Second, trends over time that depict changes in productivity must be considered. For instance, Fig. 2.1 illustrates the productivity index of two plants producing the same unit of output. While the absolute productivity of plant A is higher than the one of plant B in every point of time, the trend analysis favors plant B. Third, the productivity's output value may be ambiguous, e.g., produced picture frames vs. produced cars per labor hour, and, thus, may lead to unfair or infeasible comparisons.

This work builds upon the economic multifactor productivity index and interprets it for HPC. The constructed metric is mathematically comprehensible and can depict trends over (system life)time to compare changes in HPC setups.

2.2. Cluster Procurement in Germany

The German HPC landscape is organized into Tier-1, Tier-2 and Tier-3 HPC centers [Wis15a]: Tier-1 centers are meant for capability computing where simula-

tions demand a huge amount of processors and memory. The three existing Tier-1 centers build the Gauss Centre for Supercomputing (GCS) [Gau17b]: the Jülich Supercomputing Centre (JSC), the Leibniz Rechenzentrum (LRZ) near Munich and the Höchstleistungsrechenzentrum Stuttgart (HLRS). Tier-2 centers serve for capacity computing with compute-power for running highly-complex applications or capability tests. The 19 Tier-2 systems in Germany are organized in the Gauss Alliance [Gau17a]. Tier-3 systems provide compute power that is suitable for most scientific applications.

Funding structures differ across the three tiers. Since RWTH Aachen University belongs to the Gauss Alliance, I focus on Tier-2 systems in the following. Tier-2 centers get their funding half-half by the federal republic and the responsible federal state. The advisory council on scientific matters provides an assessment of corresponding proposals based on recommendations provided by the German Science Foundation (DFG) [Ger16]. Grants from these funding sources typically allow procurement of new cluster hardware within a fixed budget. Due to ever increasing complexity and expenses in HPC, German HPC Centers argue to have operational and development costs as part of this cost perspective [ZKI13]. The advisory council [Wis15a] and the DFG [Ger16] also recommend to include operational costs, e.g. energy expenses, and brainware (also referred to as *methodological expertise*) into the TCO calculation. At the same time, funding sources expect high benefits for researchers who exploit the purchased hardware. Thus, HPC centers are extrinsically (and intrinsically) motivated to predict the most productive environment for their researchers and customers within the given budget.

Nowadays, numerous parallel hardware architectures are on the market that differ considerably in, e.g., their number and capability of compute cores, network and bus topology, memory design, energy efficiency or development environment like supported programming models, compilers and tools. To predict the best setup, different systems can be compared by their productivity that incorporates all parameters mentioned above. Figure 2.2a illustrates exemplary productivity curves of two systems as function of the given investment. Here, system B suffers from dominating one-time costs (e.g., acquisition or porting costs) at small investments with low productivity compared to system A. However, higher investments enable to purchase more hardware: higher performance gains on system B weigh up the one-time cost hurdle. We discuss similar real-world use cases in [WaMM13].

HPC managers further pursue the question of when to invest in novel hardware. Although today's funding structure only allows this kind of optimization within the given external conditions, future directions may give more flexibility. One optimization bases on the comparison between spending allocated budget at once or procuring hardware in two phases. On the one hand, a single cluster procurement satisfies the researchers' compute needs in an one-time installation effort without suffering from great uncertainties in hardware design. On the other hand,

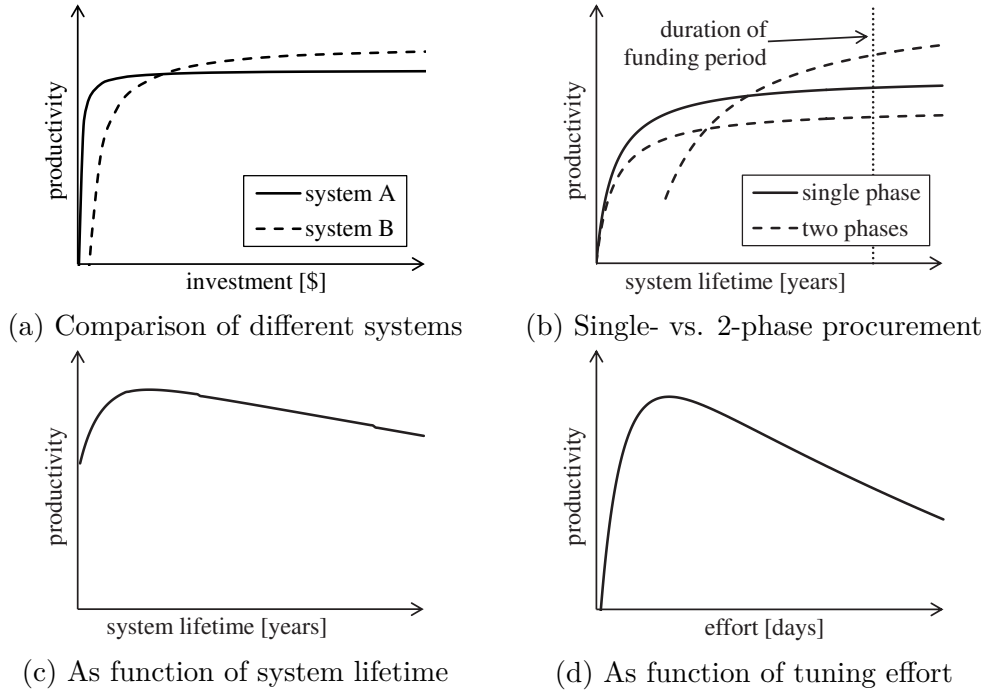


Figure 2.2.: Exemplary productivity use cases in HPC.

distributing the budget across two procurement phases offers the chance to benefit from newest technology advances in the second phase. For instance, future hardware architectures promise more performance and energy efficiency per € and estimations of the productivity metric can take assumptions like Amdahl’s Law into account. Figure 2.2b shows exemplary productivity differences of a single phase and two-phase cluster procurement over time. Here, the two-phase acquisition is beneficial at the end of a given funding period. Quantitative investigations of this use case can be found in our work Wienke et al. [WlaMM15].

Investigating the end of the lifetime of an HPC system, HPC managers and users might intuitively think of mean time to failure (MTTF) of hardware components and corresponding maintenance contracts. Modeling the productivity of a system over its lifetime can reveal further or new insights. The examination of the turnaround in the productivity curve in Figure 2.2c reveals that operational costs, e.g., software (licensing) expenses, are decisive for the length of productive employment. We describe more details of this use study in Wienke et al. [WlaMM15].

2.3. Developer’s Effort Optimization

Besides its usability for HPC managers, the productivity figure of merit can additionally be reduced to a user’s perspective and support decision making on this

developer level. For instance, it helps users to optimize their HPC development efforts. Here and in the remainder of this work, (HPC) development effort is defined as development time spent for porting, parallelizing or tuning simulation applications for HPC systems and, hence, embraces so-called HPC activities (full definition in Sect. 8.1.1). Figure 2.2d illustrates productivity as function of effort spent by incorporating the tradeoff between development time (input) and performance improvements (output). Assuming a Pareto distribution for this relationship, e.g., representing the 80-20 rule, labor costs exceed benefits from performance gains after a certain point in development time. Correspondingly, the maximum in productivity (as function of effort spent) represents the time to stop tuning. More evaluations have been performed and discussed in our publication [WIaMM15].

3. Related Work

It has been a long tradition to assess HPC systems by their peak performance. This is especially evident in the Top500 [TOP17] list that started back in 1993 and that is based on the High Performance Linpack (HPL) benchmark measuring achievable peak floating-point operations per second (Flop/s). This Flop/s metric and the corresponding ranking within the Top500 scores in politics and funding agencies [Nee14]. It is also referred to as “perhaps most-well known measure” for HPC leadership between countries by the Information Technology and Innovation Foundation [EA16]. For example, China’s government favors to power the #1 system in the Top500 since 2013 [Fel17]. Apon et al. [AAD⁺10] even show a statistical correlation of U.S. National Science Foundation (NSF) funding to the appearance within the Top500 list covering 204 institutions with Top500 entries from the Carnegie Foundation list. However, its usefulness for decision making in HPC procurement and other related investments is strongly debated. Its unsuitability is especially due to numerous HPC applications being bound to memory bandwidth or even I/O instead of sheer Flop/s power. The Flop/s metric is also restricted because not all given floating-point operations in an application may be relevant or necessary (often referred to as *Macho Flop/s*) or certain algorithmic changes that improve runtime and yield implementation effort do not even affect the (kernel’s) Flop/s number.

As extension of the Flop/s metric, the acquisition expenses have been included for the comparison of systems, yielding the metric Flop/s per dollar. Recently, the discussion on the budget factor picked up speed by the massively-increasing investments needed for energy of today’s HPC systems. Extrapolating power consumption of the Top500 #1 system in 2016 *Sunway TaihuLight* to an exaflop system, while assuming 1 Mio. € is spent per MW, this system would cost 165 Mio. € for energy per year. Therefore, the DOE constrains the upper power consumption of a future exaflop system to 20 MW to 40 MW [DOE14][DOE17a]. With that, system comparisons that also include energy costs have become popular and more and more factors have been attributed to the various costs and benefits of HPC machines.

For these comparison purposes, productivity is a common economic metric as shown in Sect. 2.1 and, hence, also used here to evaluate the cost effectiveness of HPC environments. Related work on productivity models can be mainly distinguished between user-based and system-based approaches (Sect. 3.2 and 3.3). In-

stead, business perspectives rather focus on monetary values as covered in Sect. 3.4. Across all approaches, I try to standardize the variable names for better legibility.

3.1. The HPCS Program

Productivity figures of merit in the HPC area have been almost exclusively studied in the context of the HPCS program [DGH⁺08] of the U.S. Defense Advanced Research Projects Agency (DARPA). The HPCS program was initiated in 2002 and aimed at developing economically-viable high-productivity computing systems by 2010. Its goals covered concepts for innovative technologies, prototypes and research with respect to the HPCS productivity factors: performance, programmability, portability and robustness. An HPCS productivity team that was funded from 2003 to 2006 and led by Jeremy Kepner conducted research in performance benchmarking, system architecture modeling, productivity workflows/metrics and new languages [DGH⁺08, p. 18]. Their development of tools targeted at the comparison of current HPC systems to those being developed for HPCS. The HPCS team studied productivity as time-to-solution that encapsulated execution time and development time. While I discuss their works on software development time in Part C, I will cover their productivity models (and similar works) in the following. The works have been mainly published in [DD04][Kep06a][Kep06b] and Dongarra et al. give their overview of the corresponding works in [DGH⁺08, pp. 63].

3.2. User Perspective

Productivity models that take a user perspective focus on a specific (single) application that one or more users develop, test, debug, analyze and execute on an HPC system. Thus, they account for development effort and performance (only) where development effort is usually either source lines of code or development time.

3.2.1. Relative Development Time Productivity

The HPCS program initially evaluated productivity from an end-user perspective [FBHK05][FKB05][FBHK06][DGH⁺08, pp. 50] that is based on relative performance and relative effort. Relative performance is captured as speedup defined as serial runtime divided by the parallel runtime of an optimized implementation. Effort is expressed as code size in terms of source lines of code (SLOC) which are distinguished between total SLOC and effective SLOC with the latter excluding boilerplate SLOC. A SLOC ratio was setup with SLOC Γ_0 of the serial code version

as numerator and SLOC Γ_L of the parallel optimized implementation as denominator: Γ_0/Γ_L . Then, the *relative development time productivity (RDTP)* metric is defined as speedup divided by relative effort given by the relative code size. This metric has been applied to seven different classroom experiments, to codes from the NAS Parallel Benchmark (NPB) Suite and to codes from the HPC Challenge (HPCC) Benchmark Suite evaluating the relative productivity of OpenMP, MPI and pMatlab/Matlab*P. Hollingsworth et al. [HZH⁺05] enhance this simple productivity approach. They derive their figure of merit from an absolute productivity index that has been setup as utility over cost, but focus on programmer productivity on a project level. Thus, they account for the cost of producing one program utility unit, the program size in program utility units, the (relative) cost of the parallel program and its speedup. Since they do not know the cost to produce one program utility unit, they also come down to a productivity metric defined as (relative) speedup over relative effort. They specify speedup based on a (at best serial) reference version, and further incorporate a factor r that shall account for the total lifetime execution behavior of the program. I use the terminology introduced by Kennedy et al. [KKS04] to ease the comparison of different approaches. Here, $I(P)$ specifies the implementation effort for a problem P and $E(P)$ the execution time of that program. The reference version is referred to by P_0 whereas the optimized implementation is represented by P_L . Hollingsworth et al. express productivity Ψ as:

$$\Psi = \frac{\text{relative speedup}}{\text{relative effort}} = \frac{\frac{E(P_0)}{E(P_L)}}{\frac{I(P_L)+rI(P_0)}{I(P_0)+rI(P_L)}} \quad (3.1)$$

where r abstractly reflects the number of executions of the program by a given range of $[0; 1]$ and, hence, eliminates the emphasis on a single program run. Hollingsworth et al. apply their metric to 17 data sets gathered in classroom experiments using 5 projects with effort log data and 12 projects with SLOC counts.

From my view, this RDTP approach has the following disadvantages: First, it covers productivity only relatively to some reference version where especially the parallel execution time is not well defined: running an OpenMP application on 16 instead of 8 threads without any additional effort will automatically increase productivity. Second, this relative productivity captures programmer productivity only and does not account for a cluster perspective of an HPC center by neglecting, e.g., capital and operational costs. Further, it is limited to compare single projects only and does not incorporate a system's job mix. Third, its application has mainly been shown using SLOC, and as comparison of different parallel programming models. Fourth, the physical reality expressed by factor r is unclear. My productivity metric introduced in Sect. 4 lifts these disadvantages by establishing a figure of merit that is based on absolute numbers and the HPC center's perspective while still being able to get reduced to a programmer productivity metric. In

addition, the productivity index proposed in my work captures single and multi job environments, embraces real software development efforts (instead of SLOC) and adheres to physically-real conditions.

3.2.2. Relative Power & Relative Efficiency

Similar to the user-based RDTP by Hollingsworth et al., Kennedy et al. [KKS04] investigate implementation time $I(P)$ and execution time $E(P)$ of a problem P and combine them to *time to solution* $T(P)$ that is targeted to get minimized for better productivity:

$$T(P) = I(P) + rE(P) \quad (3.2)$$

where r is an integer number that represents the importance of decreasing execution time versus implementation time for problem P . From that, they derive relative power ρ and relative efficiency ε of a new programming language interface P_L in comparison to some reference problem P_0 :

$$\rho_L = \frac{I(P_0)}{I(P_L)} \quad \text{and} \quad \varepsilon_L = \frac{E(P_0)}{E(P_L)}. \quad (3.3)$$

While they emphasize that a graphical presentation of the productivity for comparison of different programming interfaces with respect to efficiency and power is the most valuable approach, they also define a single scalar metric as relative productivity Ψ :

$$\begin{aligned} \Psi &= \frac{T(P_0)}{T(P_L)} = \frac{\rho_L I(P_L) + \varepsilon_L r E(P_L)}{I(P_L) + r E(P_L)} \\ &= \frac{\rho_L + \varepsilon_L X}{1 + X} \end{aligned} \quad (3.4)$$

with $X = rE(P_L)/I(P_L)$ and the corresponding break-even point X_{even} at which the productivities of the new and the reference programming interface equal as $X_{even} = (\rho - 1)/(1 - \varepsilon)$.

Thus, to estimate the productivity of P_L over P_0 , the relative power ρ and efficiency ε must be predicted. Kennedy et al. propose direct measurements of execution time for ε and using expert opinions (aggregated into a cumulative distribution function if needed) for ρ . To further eliminate the problem-specific character of power and efficiency, they propose taking the average (or median) of execution and implementation times for a set of standard benchmarks. Since the authors focus on the user's time component, the cost factor is just briefly touched.

Comparing the authors' work to my approach, they also incorporate development time (instead of SLOC) into the productivity metric. However, they only define relative productivity to compare different programming interfaces and, thus,

cannot serve HPC procurement processes. While the authors give details on how to measure power and efficiency, they do not describe ways to estimate these values beforehand. In addition, the factor r is still a subjective parameter that is difficult to predict either. In contrast, I suggest prediction possibilities for all parameters used in my productivity metric, as well as, base the weighting between implementation and execution time on real values by looking on the number of applications runs over the system's lifetime including total costs.

3.2.3. Response Time Oriented Productivity

Sterling and Dekate [SD08] define a more general productivity framework Ψ by

$$\Psi = \frac{U}{C \times T} \quad (3.5)$$

where T represents a time parameter, U the utility and C the cost. From that, they derive a system-based model (covered in Sect. 3.3.2) and a user-based model. Their user-based model assumes a single-application workflow with focus on “response time” capturing development time T_D , initialization and setup time T_{Qi} for an application run i and execution time T_{Ri} for the same run i over a total of K runs as defined in Equ. (3.6). Correspondingly, the cost factor captures the application's development costs C_D , its execution cost C_{Ri} over K runs and the setup cost C_{Qi} :

$$T = T_D + \sum_i^K (T_{Qi} + T_{Ri}) \quad \text{and} \quad C = C_D + \sum_i^K (C_{Qi} + C_{Ri}) \quad (3.6)$$

The single cost components are further defined by the required time, the cost per time c (e.g., € per person-hour) and the average number of people involved n (if applicable), so that exemplary for C_D holds:

$$C_D = c_d \times n_d \times T_D \quad (3.7)$$

In this user-based model, the utility U is defined as useful work W performed over K runs: $U = W = \sum_i^K w_i$ with w_i the work performed in application run i , i.e., the worth of the result of run i . Sterling and Dekate further translate W to average efficiency measures in their work. Since this definition is also part of their system-based model, I cover this in Sect. 3.3.2. In [Ste04], Sterling further refines the overall development time T_D depending on the user's programming workflow: The programming effort g is denoted per stage j where, e.g., $j \in (\text{design, code, test}) \times (\text{new, port, reuse}) \times (\text{serial, parallel})$. Each stage j is expressed as the programming time per code size unit for this stage g_j and the fraction of the application p_j that goes through this stage. The code size itself is given by Γ in arbitrary units (usually SLOC):

$$T_D = \Gamma \times g \quad \text{with} \quad g = \sum_j^{N_{stages}} g_j \cdot p_j. \quad (3.8)$$

While this response time oriented model covers the user perspective of productivity, an informed decision for cluster procurements can only be conducted on a system-wide level that the authors approach in Sect. 3.3.2 and that is considered in my productivity metric. Especially, the single-application character of their model is a very simplistic approach. For example, scientist groups may have their own financial resources and wish to use their model but may fail due to a given multi-application landscape. Therefore, I also propose a multi-job extension for modeling productivity. Taking averages across applications [Ste04] seems also as a very simplified generalization that needs further investigation. Instead, I apply an uncertainty and sensitivity analysis to all parameters of my productivity metric (see Sect. 4.6). Especially, I show that productivity can be highly sensitive on the applications' execution time why taking averages might not suffice.

3.3. System Perspective

User-based productivity models may be extended to or redefined as system-based models that incorporate the value of an HPC system and its total ownership costs. System-based productivity metrics can be used to support the decision-making process of procuring HPC machines by enabling a comparison across these machines. This generally equals the approach that I follow in the upcoming sections. However, most previously-published works investigate productivity from a theoretical side without giving methods to estimate corresponding values or incorporating real multi-job setups.

3.3.1. Utility Theory

Recapitulating, productivity is economically defined as ratio of units of output to units of input. Snir and Bader [SB04] are first to include a utility function for productivity output that describes the preference of an HPC manager. Utility describes a time-sensitive behavior $U(P, T)$ of problem P . For example, a deadline-driven utility may be represented by a constant value u if $T \leq \text{deadline}$ and 0 otherwise. Combining such a utility-time curve with a cost-time curve of an HPC system S delivers productivity Ψ as function of time to solution T similar to Fig. 2.2c:

$$\Psi(P, S, T, U) = \frac{U(P, T)}{C(P, S, T)}. \quad (3.9)$$

Snir and Bader mention the given uncertainty in the time to solution parameter and assume probability distribution functions for its stochastic behavior. Due to missing alternatives, they further suggest to use product metrics (compare Sect. 8.1) as “proxy” for time measurements of code development and tuning.

While the authors incorporate economical utility theory to HPC productivity, I find this approach unnecessarily complex for application in university HPC centers as considered in my work. Although university researchers and scientists might encounter deadline-driven utilities (e.g., project or paper deadlines), a cluster procurement process should assume that HPC users plan their simulations in time so that they do not run into time-dependent productivity issues. Furthermore, university centers usually do not run simulations whose delay means a risk to human life. Snir's and Bader's error propagation of uncertainty in time to solution to productivity is also a good approach. However, they only model time to solution as stochastic parameter and do not consider other parameters as uncertain or account for the interaction of uncertain parameters. I cover this in a global sensitivity analysis of productivity in Sect. 4.6. As it can be seen, alternatives to traditional software complexity metrics are needed to estimate time to solution. I elaborate on this in Part C.

3.3.2. Throughput Oriented Productivity

Contrary to the response time oriented user-based productivity model covered in Sect. 3.2.3, Sterling [Ste04] and Sterling and Dekate [SD08] also define a throughput oriented system-based model. Here, they approach the productivity output as total production of a machine during its lifetime T where the product is defined as the result R . "A result is a complete solution to a computational problem represented by a set of result data; the answers to the computational problem" [Ste04]. To account for the varying importance of results, they introduce weights R_i attributed to the i th result of overall K results that are produced over the machine's lifetime. Hence, productivity is defined as:

$$\Psi = \frac{R}{C \times T} \quad \text{with} \quad R = \sum_i^K R_i. \quad (3.10)$$

Time and costs are adapted for the needs of a system-wide model: total lifetime T includes the time to produce the results $T_R = \sum_i^K T_i$, overhead time T_V and downtime T_Q . Total costs cover machine costs C_M , operational costs C_O , and development costs C_D as the sum of software construction costs per result i :

$$T = T_R + T_V + T_Q \quad \text{with} \quad T_R = \sum_i^K T_i, \quad (3.11)$$

$$C = C_D + C_M + C_O \quad \text{with} \quad C_D = \sum_i^K C_{Di}. \quad (3.12)$$

The work model that Sterling has also applied to his user-based productivity can be employed for a system-wide metric as well. Here, the useful work W represents the

number of basic operations that are performed in an application algorithm so that $R = W = S \times E \times A$ with S the peak performance of the machine (e.g., maximum number of operations, memory bandwidth or lattice updates per second), E the applications' efficiency that denotes the fraction of peak performance, and the system's availability A due to downtimes or maintenance. Then, productivity is modeled as:

$$\Psi = \frac{S \times E \times A}{C} \quad \text{with} \quad A = \frac{T_R}{T}. \quad (3.13)$$

Looking at the first approach by weighting (subjectively) application results R_i , it entails the uncertainty on how such a weighting can be accomplished and who could do it. Scientists themselves probably rate their results quite high, whereas an HPC center and manager might not be able to rate simulations across different scientific fields. Contrary, the work model with respect to efficiencies delivers well-defined numbers. However, the authors did not discuss how systems powering numerous different applications can be approached with this productivity metric. Under the assumption that different applications are appropriately measured with different metrics, such a single number may not cover them all. Therefore, I will use *runtime* as embracing metric.

3.3.3. (SK)³ Synthesis Productivity

A synthesis of the productivity models of Snir and Bader (Sect. 3.3.1), Sterling (Sect 3.3.2) and Kennedy et al. (Sec. 3.2.2) is undertaken by Kepner [Kep04a] and called the (SK)³ synthesis model. Kepner applies the approach of time-dependent utility U defined as useful work W per time. Useful work per time is represented by the peak processing speed S , the application's efficiency E and the system's availability A while the development costs C_D incorporates the developer cost per time c_d (Equ. (3.7)), the code size Γ and the programming rate r in time per code size unit (Equ. (3.8)). Finally, Kepner integrates the approach of using relative power ρ and relative efficiency ε of a programming interface (Equ. (3.3)) where he defines implementation time $I(P)$ with code size and programming rate, so that $I(P) = \Gamma \times r$. Hence, the (SK)³ productivity is defined as:

$$\Psi = \frac{U}{C} = \frac{U(T)}{C_D + C_O + C_M} = \frac{S \times E \times A}{(c_d \times \Gamma \times r) + C_O + C_M} \quad (3.14)$$

$$\Psi \equiv \frac{S \times \varepsilon \times E \times A}{C_D/\rho + C_O + C_M}. \quad (3.15)$$

Kepner discusses that SLOC do not directly match development effort needed for HPC activities, and I further detail on this in Part C. Therefore, he uses code size in arbitrary units in his formula as replacement for SLOC. He investigates

applicability of his productivity metric based on an asymptotic analysis looking theoretically on simplified ‘extreme’ cases, e.g., an HPC center that focuses on HPL runs to get into the Top500. He further applies his model for comparing MPI and OpenMP to serial code development. His sample assumptions are theoretical, e.g., MPI programming is done on half the rate of serial programming, and cover a single application only. In contrast, my productivity model has been shown to be applicable in real use cases and covers the combination of several applications.

Murphy et al. [MNV06][MNVK06] extend the (SK)³ synthesis model by explicitly distinguishing between machine-level and job-level components where a job is part of a project that includes program activities with the goal of developing an efficient application and a correct and useful result. The authors present in a fine-granular (and complicated) way their productivity parameters and express integrals over the system lifetime as averages. Here, their productivity model is depicted in a simplified way:

$$\Psi = \frac{U_M \times E_M \times (S \times E \times A)}{C_M + C_A + C_D}. \quad (3.16)$$

The parameter C_M is the machine’s lifetime cost split into one-time and annual costs. The administration cost is defined as $C_A = c_A \times n_A \times T$ with c_A the cost per person-year of support, n_A the average number of staff supporting the system and T the system lifetime. Correspondingly, the development costs are given as $C_D = c_D \times n_D \times T_D$ where T_D is the average project development time for one unique program activity including time for job submission or bookkeeping. On the job level, the factor S represents the system’s resources with typically $S \subseteq \{\text{CPU, memory, bandwidth, I/O}\}$. Each resource value gets weighted by its costs per unit of that resource and normalized to C_M . Then, efficiency E denotes how well jobs use the corresponding resources and is usually averaged over all jobs. The system’s availability A is split into single availability values per resource S due to different downtimes or failures. The machine-level components are described by utility U_M and efficiency E_M . Parameter U_M represents the success of a program activity per unit of available resources S and depends on the local HPC manager’s preferences. The authors name it as the multiplier to the peak system resources in CPU units. The efficiency of the system-wide resource allocation is defined as $E_M = E_U \times E_A$ since it consists of the effectiveness of administrators E_A to efficiently allocate resources using certain tools and the effectiveness E_U that is attributed to hardware or project failures or system configuration problems.

3.4. Business Perspective

HPC is not only an important component in research and scientific activities at system-level of university centers or national labs, but also an integral part of many

companies for manufacturing and industrial applications. Their access to HPC is essential for industrial competitiveness and science leadership [EA16, p. 14][Cou14, p. 19]. Making business cases, companies (and also some university centers) focus on monetary values. Thus, they are rather interested in benefit-cost ratios instead of cost effectiveness (see Sect. 2.1). Other common metrics are return on investment (ROI), the internal rate of return (IRR) or net present values (NPVs) that are briefly discussed in Sect. 4.5.

3.4.1. Benefit-Cost Ratio

A derivation of the (SK)³ synthesis model (Sect. 3.3.3) for industry use cases is conducted by Reuther and Tichenor [RT06]. They replace development time and costs by general costs for software C_S (e.g., costs for products by independent software vendors (ISV)). They further consider costs for training C_T , costs for administration C_A and machine costs C_M . In the numerator, they focus on (monetary) benefits as the “value of newly developed products, potential increases in market share, profits generated (or lost) using HPC systems, or the importance of the job to be completed (i.e., how much revenue or market share will the company be able to gain once this large, extremely important problem is solved)”. Hence, they define productivity as benefit-cost ratio for a one-year time period as:

$$BCR = \Psi = \frac{\sum \left(\text{profit gained or maintained by project} \right)}{C_S + C_T + C_A + C_M}. \quad (3.17)$$

They apply this metric and the internal rate of return given as $IRR = BCR - 1$ to a research laboratory and industrial production example where the latter is based on theoretical assumptions. For the first example, they used *time saved by users* as value of the HPC system and converted it into monetary units by accounting for people’s salary. The industry example is based on projects’ overall profit numbers gained by purchasing the HPC system.

3.4.2. Return on Investment

Another way to look at the economic value of HPC investment is the financial (and innovation) ROI. In 2013, a pilot study [JCD13] funded by the DOE and conducted by the HPC group of the International Data Corporation (IDC) — now known as Hyperion Research — evaluates 208 case studies with respect to revenue (similar to the gross domestic product (GDP)), profit or cost savings per invested dollar and created jobs created. Until 2016, IDC increased their collected data sets to 673 examples from which 148 reported on financial ROI and 525 on innovation return on research [JCSM16]. Their results on financial ROI can be summarized as follows:

- \$515 on average in revenue per \$ of HPC invested
- \$52 on average of profits (or cost savings) per dollar of HPC invested
- 2,335 jobs created across financial ROI projects
- \$270 K cost on average of HPC investment per job created

A similar (but switched) perspective on jobs created by HPC investment is taken by Bischof et al. [BaMI12]. They show that investing into people spending effort on tuning HPC applications can result into significant savings in HPC total ownership costs. In their setup at RWTH Aachen University, the employment of 1.5 full-time equivalents (FTEs) that achieve 10 % performance improvement on the HPC center’s 15 top projects yield 185,000€ savings in one year.

Thota et al. [TFW⁺16] compute the ROI of their locally-owned *Big Red II* system at Indiana University, USA. They weight grant incomes over their investments into Big Red II per year: At Big Red’s half life span, the grant income linked to the system across all university departments aggregates to roughly \$40 M and is projected to be at \$90 M over the total life span of 5 years. In contrast, they estimate the costs for staff and support at ~\$15 M over the system’s lifetime. Breaking it down to the facility and administration funds, the grants income equals approximately \$6 M per year which is twice the amount of Indiana’s annual investment.

Moving from locally-owned resources to the centralized approach followed by the Extreme Science and Engineering Discovery Environment (XSEDE), Stewart et al. [SRK⁺15] discuss value added by the centralization of national research computing access over the alternatives of having two or four independent national centers, e.g., in security effectiveness or national outreach. They further collect the value to the XSEDE service providers by interviews in terms of numbers of FTEs that would be needed to offer the same services locally. The benefit accounts to \$11,657,500 per year as total over seven service providers, whereas the cost avoidance of XSEDE only comes up to \$18,418,250 per year compared to four independent national centers. They argue that scientific benefits may be encountered only with a time lag of years or decades, e.g., simulations to avoid stock market crashes, and also find that the end users’ value have not been incorporated into the equation so far. Therefore, they qualitatively assess the return on investment by distinguishing costs per service provider and end user. Thus, for a $ROI > 1$, XSEDE’s end user value must be above \$460 per year.

3.5. Other Approaches

The appraisal of HPC systems has typically not been carried out using sheer productivity models, but other approaches that also have relevance in the comparison of HPC setups and procurement processes. HPC procurement processes

are not well documented and the only available summary is covered in two SC16 tutorials [JHT16][JT16]. To give some alternatives, I briefly introduce further approaches to compare HPC environments.

3.5.1. Benchmark Suites

Benchmarks have a long tradition in the procurement and commissioning process of HPC systems [JHT16]. They have been used to compare tenders of different vendors who bid results for these benchmarks and, hence, are part of corresponding acceptance tests. While HPL is still the most famous benchmark, today's acceptance tests also demand good results in HPC Conjugate Gradients (HPCG) [DH13] or Standard Performance Evaluation Corporation (SPEC) benchmarks [Sta17]. Especially, SPEC performance results are usually closer to real-world setups than HPL results since SPEC follows the concept of providing application-based benchmarks that represent the behavior of real-world simulation codes. For HPC purposes, SPEC benchmark suites come in the flavors of *OMP*, *MPI* and *ACCEL*. I engage in the development of the SPEC ACCEL suite since 2013 [JBC⁺14][JHJ⁺16][JHC⁺17] which accounts for the emergence of accelerator devices and exploits the industry standards OpenACC, OpenCL and OpenMP for accelerator devices. Nevertheless, SPEC application benchmarks may not be (fully) representative for the center's own workload. Therefore, in addition, center-specific job mixes are often used in a procurement process that illustrate several real applications of the site (usually, up to 25 applications are chosen [Jon16]). Due to confidentiality reasons or other limiters, these job mixes may also only reflect a small fraction of actual applications.

Faulk et al. [FGJ⁺04a][FGJ⁺04b] take a different approach by envisioning a *productivity benchmark suite (PBS)* that targets at multidimensional productivity benchmarks for properties like availability, reliability, repeatability, portability, reusability, maintainability. Its key components are canonical workflows [Kep04b] that characterize the development and execution process, purpose-based benchmarks [Gus04] that embody development and platform challenges of real applications in reduced size, non-functional requirements such as administration or runtime behavior, the *characteristic value function* and metrics and tools to measure productivity. The characteristic value function is represented either as vector or sum of completion metrics P_i and relative value weights ν_i for a property i (over a total of n properties):

$$V = (\nu_1 P_1, \nu_2 P_2, \dots, \nu_n P_n) \quad (3.18)$$

$$V = \nu_1 P_1 + \nu_2 P_2 + \dots + \nu_n P_n \quad (3.19)$$

$$\Psi = V/W \quad (3.20)$$

Here, P_i measures the degree of how close the property i is to the HPC manager's goals. It can be calibrated to $P_i = 1$ for complete agreement and $P_i = 0$ for complete disagreement. The relative value weight ν_i describes the importance of property i with respect to the other properties, e.g., as percentage. From that, Faulk et al. derive relative productivity Ψ as characteristic value divided by the work W required to produce it (Equ. (3.20)). Although, this approach seems to comprehensively embrace all interesting properties of an HPC system and the corresponding application development process, its realization is a huge challenge. While the authors discuss how the property of maintainability could be measured this way, the implementation of other soft properties are not reviewed. Since there are no further works on the PBS, it seems that the authors' work stays visionary.

3.5.2. Research Competitiveness

At the level of academic institutions, another important metric for comparison among departments or universities has been present for a long time, i.e., the amount of external funding and the amount of publications. Apon et al. [AAD⁺10] investigate the relationship between changes in these research competitiveness metrics and investments in HPC systems where the latter is represented by entries on the Top500 list. They conduct a correlation and regression analysis on 204 academic institutions that appeared in the Top500 from 1993 to 2007 and show effects of the Top500 list count on NSF funding levels and contemporaneous increase in the number of published work. In fact, one point increase in overall Top500 ranking score relates to an average increase in NSF funding of \$2,419,682 and an average increase of 60 publications. Further, Apon et al. [ANPW15] apply nonparametric efficiency estimators to test the effect of locally-available HPC resources on universities' technical efficiency in producing research output distinguished by discipline. Local HPC capability is tracked by the center's appearance on the Top500 list from 2000 to 2006, whereas the efficiency estimator includes as inputs the total number of faculty (in FTEs) and the average graduate record examination scores of the department's incoming graduate students and as outputs the total number of publications for the academic year, and the number of Ph.D. degrees awarded. They demonstrate that locally-available HPC resources increase the technical efficiency of research competitiveness in Chemistry, Civil Engineering, Physics, and History.

To increase the understanding of grants income and publication data linked to (local) HPC resources, the NSF has recently funded the extension of the open-source utilization tool XDMoD (XD Metrics on Demand) to develop the value analytics module (XDMoD-VA) that supports compilation of corresponding metrics [TFW⁺16][Ind17].

3.5.3. Cloud Computing

While Cloud Computing is not a focus of my productivity modeling and evaluations, numerous works exist that compare HPC resources available in the cloud among each other or to on-premise HPC systems with respect to performance, cost or cost effectiveness [CHS10][GM11][NB09][DWZ15][DaMW⁺13][DaMW⁺14]. For example, in Ding et al. [DWZ15], we compare static versus dynamic work scheduling with MPI on Azure cloud nodes and compute costs per application run (compare Sect. 4.5). The broad cost discussion is originated in the nature of cloud computing providing on-demand cost-effective alternatives to locally-available HPC environments. Furthermore, the pricing model of cloud resources that usually covers costs for computation and data transfer makes it especially easy to predict the corresponding total costs for an application run. Nevertheless, the productivity output metric of corresponding HPC application runs suffers from the same challenges as the productivity output of on-premise HPC systems.

4. Modeling Productivity in HPC

As motivated in Sect. 2, the productivity metric can practically be applied in HPC setups and increases in importance with upcoming expenditure and funding structures [Wis15a]. Nevertheless the definition of productivity in HPC is challenging and also controversial as seen in Sect. 3. I introduce a productivity model with predictive power that delivers a quantifiable metric for real-world scenarios and whose composition is easily understandable by HPC managers, decision-makers and developers to affect their behavior and enable the analysis of impact factors.

In Sections 4.1 to 4.3, I present the model definition of my productivity index and illustrate its applicability to real-world setups. I further discuss in-depth the model's components and their predictability in Sect. 4.4. In Sect. 4.5, I introduce further metrics that can be derived from my productivity approach. Section 4.6 covers an uncertainty and sensitivity analysis of my productivity model and its parameters. Finally, Sect. 4.7 illustrates some tools to support the modeling of productivity of HPC centers.

The work presented in this chapter is partially based on our previously-published research by Wienke et al. [WIaMM15]. There, I have introduced a productivity model that incorporates one-time and operational costs (including manpower) and large-scale performance models while focusing on single application setups. The paper's productivity model was further parameterized resulted from discussions with the co-authors. Other work that directed towards the productivity model introduced in this work has been published in Wienke et al. [WaMM13].

Moreover, the model relies on experiences and data gathered by managing and monitoring the RWTH's Compute Cluster from 2011 to 2016. Data includes cluster usage statistics collected by Linux accounting and the batch scheduler. Further details are taken from project proposals that scientists submitted for the application of compute time at RWTH Aachen University and its JARA-HPC partition [JAR17d][JAR17a]. Such a proposed project often comprises few different application setups and is worked on by several research employees. In addition, experiences and data are based on the proposal and tendering process for the RWTH cluster CLAIX that was taken in production in November 2016.

As part of project proposals and as integral component of any cluster and charging statistic, the consumption of core hours is commonly recorded and evaluated. The number of core hours describes the capacity that a simulation consumes: A

simulation that takes 24 cores for 2 hours consumes 48 core hours. Although the characteristics of cores are nonuniform (e.g. frequency, design), this measure has been widely accepted. Nevertheless, I reformulate core hours as the unit *node hours*. Compute-node hours account for differences in architectural design, e.g. GPU nodes can be more easily compared to CPU nodes than GPU cores to CPU cores. If needed, node hours can still be reckoned back to core hours depending on the cores per node.

4.1. Definition of Productivity

The broad definition of the basic economic productivity allows and forces a more detailed definition for particular business cases. This also holds for HPC setups. However, university HPC centers do not directly sell products and, thus, their output is not well defined. Instead, they support advances in science and research as main contribution. The huge challenge lies in the numerical quantification of science and research with their intangible character. To tackle this problem, I define the *number of application runs* that an HPC center of size n (number of compute nodes) can perform over the system's lifetime τ as its value:

$$\text{output} := \sum_i \text{application runs} = \sum_i r_{app,i}(n, \tau) \quad (4.1)$$

where $r_{app,i}$ represents the number of runs for application i . The number of application runs can easily be quantified and also predicted by applying performance models to the set of applications. They strongly depend on the applications' run-times, i.e., the application performance running on n compute nodes of the HPC center. More details are described in Sect. 4.2.

The input costs of the productivity of an HPC center can be straightforward defined. My approach sums up all substantial one-time costs C^{ot} and annual costs C^{pa} that contribute to the center's total cost of ownership. This includes acquisition costs, maintenance costs, energy costs, software costs, labor employment costs or labor porting and tuning costs. While manpower expenses has not been considered as important cost factor in the area of HPC for a long time, the rise of energy costs of large HPC clusters and corresponding cooling has also sharpened the awareness of growing manpower efforts that accompany the increasingly complex hardware developments. This goes to the extent that recently the Association of German University Computing Centers (ZKI) and the German advisory council on scientific matters have recommended including brainware and methodological expertise into the governmental funding structure. Thus, the productivity input is given as the TCO of an HPC center of size n over the system's lifetime τ where software development efforts are part of the one-time costs C^{ot} :

$$\text{input} := \text{TCO}(n, \tau) = C^{ot}(n) + C^{pa}(n) \cdot \tau. \quad (4.2)$$

Although this definition seems simple and the metric easily quantifiable, the difficulty arises at the assignment of numerical values to the TCO parameters. In addition, some TCO components are tricky to predict. These issues are further discussed in Sect. 4.3 and Part B.

Composing value and cost of an HPC center, my definition of productivity Ψ is as follows:

$$\Psi(n, \tau) = \frac{\sum r_{app,i}(n, \tau)}{\text{TCO}(n, \tau)}. \quad (4.3)$$

The defined productivity in Equ. (4.3) tackles most inherent challenges of productivity indices described in Sect. 2.1. First, my productivity metric is comprehensible without economical or statistical background. This increases the willingness of HPC developers, HPC managers and even governmental staff who make decisions on HPC investments to affect their behavior by being able to interpret the metric and its impact factors. Second, this productivity metric features comparisons of different setups within an HPC center, such as alternatives in purchasing parallel hardware architectures or applying parallel programming models. Although a comparison across different HPC centers can theoretically be imagined, in real world, it is unlikely due to the enormously varying application sets that run on different HPC sites. This fact represents the only drawback of this metric. Third, this productivity index also exploits trends over time of an HPC center by directly including the system's lifetime as an independent parameter. Forth, the ambiguity of the productivity's output is removed by concretely defining it and by not enveloping it by mathematical utility functions.

Finally, the productivity measure of an HPC center has the most value if it can be applied apriori to procuring and installing a new HPC setup. This follows the actual objective of cost effectiveness, namely, to *predict* the best alternative. Besides its comprehensiveness, the productivity index given in Equ. (4.3) is reasonably predictable by building the overall estimation modular from the single components estimations. For most components this is feasible by using models (e.g., performance or energy models) or by talking to vendors (e.g., acquisition or licenses). The predictability of other components (e.g., effort needed for HPC application development) is more challenging and, hence, is a main part of this work (compare Part C). Either way, early estimations always come with certain uncertainties that are addressed in Sect. 4.4 and analyzed in-depth in Sect. 4.6.

While I have presented a rough definition of a productivity index for HPC so far, I clearly define productivity output and input metrics in the following sections. Here, I differentiate between application-dependent and system-dependent components. Runtime and scaling behavior, consumed power and effort needed to parallelize, port or tune an application belong to the application-dependent data. All other components are defined to be part of system-dependent data, e.g. hardware acquisition costs, software licenses costs or system availability. This dis-

tion follows the idea for cluster procurements that application-dependent data is provided directly or through interviews with application developers or their superiors (e.g., joint applicants for the cluster proposal), while the remaining data comes from the HPC center's managers, e.g., collected in interviews with vendors or from previous experiences.

4.2. Value: Number of Application Runs

The output and numerator of the productivity index in Equ. (4.3) is defined as the number of application runs that are executed on a supercomputer over its lifetime. This definition is based on the supposition that each *scientific output* of an application delivers insights into the corresponding scientific field, no matter whether the result represents a scientific success or a scientific nonachievement. For instance, a simulated parameter set can optimize certain material properties in engineering or it may exclude a material from further investigations, respectively. As inherent in the description above, an *application* belongs semantically to a simulation software package. Within this simulation software, several experiments may be simulated, e.g. by applying a parameter study or by combining different simulation stages into one framework. Here, one *application run* is defined as simulating one such experiment. If in doubt, the domain scientist may decide on which experiments deliver scientific output. From this definition that is based on scientific output, it is naturally to focus on production runs including pre and post processing. Consequently, test runs or debugging sessions are excluded from this counting rule for the moment. However, since debugging and testing activities can consume extensive effort, I acknowledge these in the (development) cost factor of productivity (see Sect. 8.1). From a computing center's perspective, an application run can be hidden behind any kind of batch job: serial, parallel or array jobs. In addition, it is independent of the hardware architecture and may employed on x86 compute nodes, GPU nodes, etc., or even in hybrid mode (e.g. host and GPU).

A typical cluster is exploited by numerous different applications representing the cluster's job mix. In a first step, I abstract this job mix to a single application perspective for simplicity. Secondly, I compose the number of single-application runs to a comprehensive and applicable job-mix definition.

4.2.1. Single-Application Perspective

The single-application abstraction allows focusing on the important components of application runs first before getting into the complexity of multiple applications. This abstraction also covers real use cases. For instance, RWTH Aachen university institutes with their own budget are interested in purchasing hardware,

and computing the productivity for their own single research application. They could integrate their own compute nodes as part of the RWTH Compute Cluster using the integrative hosting service offered by the IT Center [ITC16]. In this way, hardware and software are managed by the same set of tools as the rest of the cluster and administrative costs are minimal. Thus, the institute’s cost-effectiveness analysis is based on one application that is running 24/7 on their own compute partition.

In formulas, the number of runs for one application r_{app} can be computed by dividing the overall time that the system is available by the application’s runtime t_{app} :

$$r_{app}(n, \tau) = \frac{\alpha \cdot \tau}{t_{app}(n)} \quad (4.4)$$

where the parameter α describes the system availability rate in percent and addresses additional scheduling delays, maintenance periods or unreliability of the system. It could be setup in analogy to previously-experienced availabilities from on-premise clusters, e.g. to 95 %. Nevertheless, the main impact on the number of application runs is the application’s performance that is given by its scaling behavior $t_{app}(n)$. I set it up as function of the number of nodes (instead of number of cores) to ensure a consistent view on different kind of architectures that do not necessarily count cores in the same way. For example, this allows to define parallel scaling uniformly across CPU and GPU nodes. To predict the application’s runtime behavior, performance models¹ can be used. Thus, $t_{app}(n)$ can be assumed to be as simple as following Amdahl’s Law or very application-specific by relying on caching and memory behavior and including further performance counters values. Using Amdahl’s Law as example, HPC centers can easily incorporate future technological progress into the formula by applying abstract assumptions that performance may predictably be increased by a factor two to three when going to a novel or other architecture, e.g., GPUs. My case study covered in Part D exemplifies this approach. When modeling the application’s parallel scaling behavior, the number of employed nodes n can be decisive since the application might not efficiently leverage a whole cluster of maybe thousands of nodes. In such case, a threshold n_{scale} that defines the maximum number of nodes on which the application scales acceptably can be used instead. The n acquired nodes are then divided into multiple partitions of size n_{scale} and eventually of a remainder of size n_{rem} :

$$n = \lfloor n/n_{scale} \rfloor \cdot n_{scale} + n_{rem} \quad \text{with} \quad n_{rem} = n - \lfloor n/n_{scale} \rfloor \cdot n_{scale} \quad (4.5)$$

$$r_{app}(n, \tau) = \alpha \cdot \tau \cdot \left(\frac{\lfloor n/n_{scale} \rfloor}{t_{app}(n_{scale})} + \frac{1}{t_{app}(n_{rem})} \right). \quad (4.6)$$

The formula is also valid for single-node applications by setting $n_{scale} = 1$ and $t_{app}(n)$ proportionally to n .

¹Performance modeling in HPC is its own huge research field and therefore can and will not be extensively covered in the context of this work.

My previous definitions focused on strong scaling aspects of an application by varying the number of nodes only. However, especially distributed-memory application runs are employed to simulate increasingly large data sets, e.g., representing higher resolutions, by performing domain decompositions. Therefore, running on a high number of nodes has a benefit that is not always expressed in the number of application runs. The only way to account for this is the introduction of a weighting quality factor $q_{app}(n)$ that can take different forms depending on the specific case:

$$r_{app}(n, \tau) = \alpha \cdot \tau \cdot \left(q_{app}(n_{scale}) \cdot \frac{\lfloor n/n_{scale} \rfloor}{t_{app}(n_{scale})} + q_{app}(n_{rem}) \cdot \frac{1}{t_{app}(n_{rem})} \right). \quad (4.7)$$

For example, $q_{app}(n)$ could take a value of 1 for executions on n_{scale} nodes and then decrease linearly when fewer nodes are used. A typical weak scaling test could yield $q_{app}(n) = n$, or the benefits of 3D data sets could be weighted by $q_{app}(n) = (n/n_{scale})^3$. I discuss the challenge to attribute weights across several applications in Sect. 4.4.

4.2.2. Composition to Job Mix

Having defined the weighted number of single-application runs, I extend this single-application view to a cluster job-mix perspective that represents the productivity's output metric:

$$\text{output} = r(n, \tau) = \sum_{i=1}^m r_{app,i}(n, \tau) \quad (4.8)$$

where m represents the number of (relevant) applications running on the super-computer.

Number of Relevant Applications

On a cluster, several hundred different applications may run per year amounting to the job mix mentioned above. For example, accounting information shows that 218 reviewed projects used the RWTH Compute Cluster in 2015. For simplicity, I assume in the following that the number of different projects equals the number of different applications, although the application number might be slightly higher. If an HPC manager was forced to collect application-dependent information for each of these hundreds of applications, a mere aggregation of all predicted application runs would be laborious and not applicable in real world. An acceptable tradeoff between accuracy and feasibility is given by a reduced application number that only accounts for the most important applications. One way of reduction is based on a typical German procurement proposal for HPC environments that

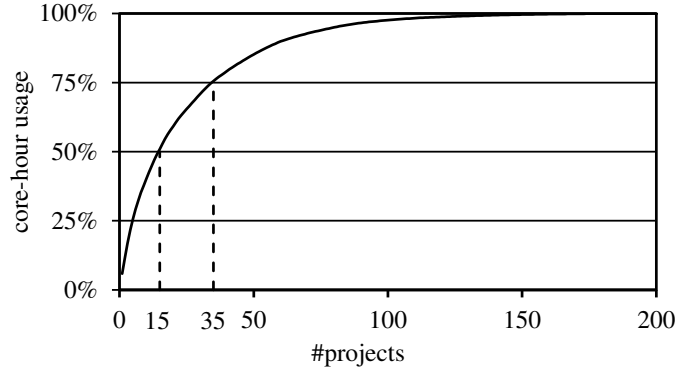


Figure 4.1.: Projects at RWTH Compute Cluster in 2015 ordered by core-h used.

lists the main and joint applicants. In 2015, the RWTH CLAIX call was written by the RWTH’s IT Center and further nine joint applicants of the university that are important representatives of their fields. These joint applicants could provide predictions for their research applications and the application number is decreased to tens of applications (instead of hundreds). Concomitant with the CLAIX call for tenders, a job mix has been set up that represents applications from the computationally most important faculties at RWTH Aachen University. Bidding companies were asked to not only deliver good performance in hardware benchmarks like HPL or Stream [McC95], but also in these relevant real-world applications. This job mix differed from the one of joint applicants due to certain restrictions such as confidentiality of source code and data sets. Thus, this job mix may also be a basis of a reduction of total number of applications running on a cluster. A third way to reduce the number of applications to a feasible set lies in the analysis of previous cluster statistics—if available. This statistical data reveals the most important applications in terms of core-hours (or node-hours). In 2015, 15 projects account for 50 % and 35 projects for 75 % of the compute load at the RWTH Compute Cluster. Figure 4.1 illustrates this distribution of projects ordered by monthly-averaged core hours used in 2015. Thus, a focus on, e.g., 75 % of the top projects reduces the application number to the order of tens. In Sect. 4.6, I analyze the sensitivity of this simplification.

Capacity-Based Distribution of Applications

For the composition of the identified relevant applications, the previously-defined single-application view in Equ. (4.4) must be extended to account for the real distribution of these applications on a cluster. In this context, I examine the system’s capacity that is given by its number of nodes multiplied by the system’s lifetime: $\text{node-h}_{\text{cluster}} = n \cdot \tau$. In addition, cluster accounting information commonly reveals the number of node hours that each project used in one month or year by aggregating all runs over the given period of time, the requested number of compute

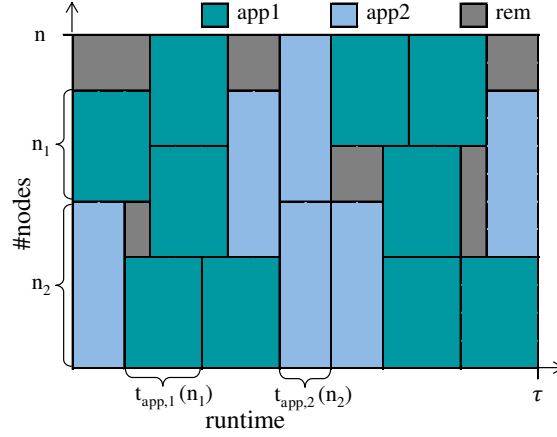


Figure 4.2.: Cluster capacity and simplified sample share of applications.

nodes and the runtime in wall clock time. Assuming that all runs of application i use the same number of resources n_i and that the system's lifetime represents the recorded period of time, then the used capacity of application i resumes to $\text{node-h}_{app,i} = n_i \cdot t_{app,i} \cdot r_{app,i}$. Thereby, the node-hour metric also abstracts from the actual order in that the different applications are sharing the cluster. A simplified sample distribution can be seen in Fig. 4.2. Here, two applications $app1$ and $app2$ mainly occupy the given cluster's node hours, i.e., 56 % and 33 %, respectively. The remaining part (in grey) is consumed by many small applications that amount to roughly 11 % of the available cluster's node hours. Deducing from the explanations above, the percentage p of totally-available node hours on a cluster is an appropriate weighting factor to account for the real distribution of applications on a cluster. Hence, the capacity-weighted number of runs for application i is defined as

$$\begin{aligned}
 \text{node-h}_{app,i} &= p_i \cdot \alpha \cdot \text{node-h}_{cluster} \\
 \Leftrightarrow t_{app,i}(n_i) \cdot n_i \cdot r_{app,i} &= p_i \cdot \alpha \cdot n \cdot \tau \\
 \Leftrightarrow r_{app,i}(n, \tau) &= p_i \cdot \frac{\alpha \cdot n \cdot \tau}{n_i \cdot t_{app,i}(n_i)}
 \end{aligned} \tag{4.9}$$

with $0 \leq p_i \leq 1$ and $\sum_i p_i \leq 1$. In real world, values for the percentages p_i can be derived by taking accounting information from previous cluster setups and scaling the corresponding capacity shares to the potential capacity of the new system while considering modeled runtime improvements in $t_{app,i}$. This assumption especially holds given the scientific review and assessment process of project proposals at German university centers where applicants must justify their required compute time and resources [Wis15a, p. 21][Gau16][ITC17a]. Thus, researchers may not thoughtlessly apply for and exploit more compute time because a new system provides more capacity. Nevertheless, advances in application algorithms and data resolution will probably yield higher compute requirements gradually over

the lifetime of the system. With this share-based capacity approach, a possible reduction to identified (relevant) applications is also still valid by adapting the overall available node hours of the new systems accordingly. Alternatively, the needed capacity in node hours can be extrapolated directly by researchers that may contribute to a cluster proposal.

From Single to Multi Jobs

The definition in Equ. (4.9) is also in line with the single-application view in Equ. (4.4) since then it holds $n_i = n$ and $p_i = 1$. Furthermore, it can be assumed that the breakdown to n_{scale} and n_{rem} in Equ. (4.6) can be ignored in this job-mix perspective because researchers supposingly aim only at efficiently-scaling application runs and remaining nodes will be leveraged by other applications. Thus, $n_{\text{scale}} = n_i$ and $n_{\text{rem}} = 0$. Considering the introduced quality weighting factor $q_{\text{app}}(n) = q_{\text{app},i}(n)$ in Equ. (4.7), the number of application runs is given by:

$$r_{\text{app},i}(n, \tau) = p_i \cdot \frac{q_{\text{app},i}(n_i) \cdot \alpha \cdot n \cdot \tau}{n_i \cdot t_{\text{app},i}(n_i)} \quad (4.10)$$

Overall, I have shown that my approach to model science as number of application runs (Equ. (4.8) and (4.10)) provides numerous benefits. The number of application runs $r(n, \tau)$ is a quantifiable metric that is able to comprehensively cover a real-world job mix, but can also be simplified to a single-application perspective. The model's components are comprehensible and composed in an intuitive way so that the overall model is understandable to affect the behavior of decision makers. Furthermore, the components' values are predictable to allow decisions in an ex ante analysis. Finally and most important for such a metric, I have illustrated that this model can be applied in real-world setups by investigating previous and upcoming cluster procurements at RWTH Aachen University.

4.3. Cost: Total Cost of Ownership

The input and denominator of the productivity index in Equ. (4.3) is defined as the total cost of ownership of an HPC center of system size n and over the lifetime τ . I briefly discuss TCO components in this section while more details are presented in Part B. "Total cost of ownership represents the cost to the owner to purchase/build, operate and maintain a data center" [WK11] and comprises numerous parameters. The parameters might slightly differ depending on the institution and perspective. Here, I continue to concentrate on an HPC center's view with focus on decision making in a procurement process. Again, I start

Table 4.1.: Possible TCO Components.

one-time costs C^{ot}	per node	$C_{HW}^{ot,n}$	HW acquisition
		$C_{IF}^{ot,n}$	Building/ infrastructure
		$C_{VE}^{ot,n}$	OS/ env. installation
	per node type	$C_{EV}^{ot,nt}$	OS/ env. installation
		$C_{DE}^{ot,nt}$	Development effort
annual costs C^{pa}	per node	$C_{HM}^{pa,n}$	HW maintenance
		$C_{IF}^{pa,n}$	Building/ infrastructure per watt
		$C_{VM}^{pa,n}$	OS/ env. maintenance
		$C_{EG}^{pa,n}$	Energy
	per node type	$C_{EM}^{pa,nt}$	OS/ env. maintenance
		$C_{SW}^{pa,nt}$	Compiler/ software
		$C_{DM}^{pa,nt}$	Application maintenance

with the single-application abstraction and then extend the approach to a multi-application perspective.

My main categorization of costs divides TCO into one-time and annual costs. A further breakdown is described on a *per node* (superscript n) and *per node type* (superscript nt) basis. This secondary categorization is motivated by a comparison of general interest of different types of architectures. For example, with the emergence of accelerators and their promising high performance per watt ratio, HPC managers needed to investigate whether benefits in runtime and annual energy costs weigh up additional administrative and programming costs in employment compared to traditional CPU-based types. Results of such a comparison are further discussed in [WaMM13] and Part D. A third, overarching categorization is between *application-dependent* and *system-dependent* parameters as discussed previously. A list of possible TCO components can be found in Tab. 4.1 and is further discussed in the following.

4.3.1. Single-Application Perspective

The total costs of ownership components highly depend on the (university) institution, the funding structure and targeted system type in occurrence, feature set and

amount. The components described here are the ones that suited most the HPC center setup at RWTH Aachen University and for which could be shown that they comprehensively describe HPC productivity and real values can be found. However, other sites may add, remove or refine certain components. Thus, these TCO components list should be seen rather as food of thought what to include in TCO computation and makes no claim of being complete.

One-Time Costs

One-time costs comprise numerous system-dependent components like initial expenses for hardware and infrastructure. The term infrastructure refers to power and cooling infrastructure, storage infrastructure and also to network components like switches and wires. Possibly, expenses for floor space or a building that holds the hardware must also be accounted for. These costs usually scale (or can be expressed as scaling factor) with the number of compute nodes. Costs for setting up the operating system (OS) and software environment, e.g., drivers, might also be done per node. However, for bigger clusters, touching every node manually is not efficient. Instead, concepts to automatically roll out updates to all nodes of a system type simultaneously are often put in place.

On a per node-type basis, labor effort is the main contribution. As system-dependent costs, manpower costs arise for the installation of an OS and the environment setup as mentioned above. For an HPC center seen as service provider, this includes the design of an operating concept, the investigation and configuration of an appropriate software and tool chain, as well as the integration into a job scheduler that is adapted to users' needs with respect to a certain node type. As application-dependent component, developing and programming efforts to port user applications to the targeted hardware type or tune these applications for the targeted hardware type are incorporated. Effort estimations involve numerous challenges that are further discussed in Part C.

All together, I define the one-time costs C^{ot} by

$$C^{ot}(n) = C^{ot,n} \cdot n + C^{ot,nt} \quad (4.11)$$

where $C^{ot,n}$ is the sum of all one-time costs per node, $C^{ot,nt}$ the sum of all one-time costs per node type and n the number of nodes that can be bought by the available budget.

Annual Costs

Annual costs aggregate expenses for maintaining an HPC system. Node-based costs of the system cover hardware maintenance expenses that are often determined as percentage of the purchase costs (e.g., 10%) and costs for maintaining

the OS and software environment. If costs for building and floor space cannot be usefully accounted as one-time costs, they may be broken down into annual building or infrastructure costs. Energy costs and energy efficiency got huge attention in recent exascale discussions and lot of research has been conducted in this area. While I leave its optimization and modeling for researchers in that field², I still incorporate basic energy models that could be easily extended to more accurate and complex ones. Energy costs are dependent on the hardware, the power usage effectiveness (PUE) of the computing center, the regional electricity cost and dominantly scales with the number of compute nodes. However, it is also dependent on the power consumption of the running application and, therefore, I categorize it as application-dependent component.

Per node type, application codes must be maintained (application-dependent costs), as well as the OS and software environment (system-dependent costs). Furthermore, compiler and other software licenses must be purchased which usually come with certain features for a given hardware node type.

Hence, the costs per anno C^{pa} are given as

$$C^{pa}(n) = C^{pa,n} \cdot n + C^{pa,nt} \quad (4.12)$$

where $C^{pa,n}$ is the sum over all annual costs per node, $C^{pa,nt}$ is the sum over all annual costs per node type and n the number of nodes.

Total Costs

Combining both cost types, I define the total costs of ownership as function of the number of nodes n and the lifetime τ of the system:

$$\begin{aligned} \text{TCO}(n, \tau) &= C^{ot} + C^{pa} \cdot \tau \\ &= (C^{ot,n} \cdot n + C^{ot,nt}) + (C^{pa,n} \cdot n + C^{pa,nt}) \cdot \tau \end{aligned} \quad (4.13)$$

Assuming that an HPC manager has a fixed budget I at their disposal, it is clear that the TCO must be lower than or equal to that budget:

$$\text{TCO}(n, \tau) \leq \text{budget } I \quad (4.14)$$

Further assuming that the lifetime τ can be estimated, Equ. (4.14) can be used to compute the number of nodes n that can be acquired with the given budget.

²Comprehensive energy and power models for HPC systems are out of scope of this work.

4.3.2. Composition to Job Mix

Enhancing the TCO model to multiple applications, I keep the system-dependent costs as they are and adapt the cost perspective from the three application-dependent components: programming effort, power consumption and application maintenance. Labor costs for development $C_{DE}^{ot,nt}$ and maintenance $C_{DM}^{pa,nt}$ are independent of the number of nodes and must be fully accounted for each application i . Thus, they are summed over all (relevant) applications m :

$$C_{DE}^{ot,nt} = \sum_{i=1}^m C_{DE,app,i}^{ot,nt} \quad (4.15)$$

$$C_{DM}^{pa,nt} = \sum_{i=1}^m C_{DM,app,i}^{pa,nt} \quad (4.16)$$

While it seems that the total costs are independent of the number of application runs $r(n, \tau)$ defined in Equ. (4.8), the development effort highly relates to the achieved or targeted application performance t_{app} . I discuss this relationship further in Part C.

The application-dependent part of the energy costs directly depends on the number of nodes n_i that an application i is running on, on the application's runtime and on the application's power consumption that is usually high for code parts running in parallel. For now, I simplify the model by assuming that the application's power consumption and wallclock time is on each node roughly the same as it might be the case for a load-balanced MPI application. Thus, I can combine costs for power consumption and runtime into $C_{EG,app,i}^{pa,n}$. With extensive infrastructure for power/energy measurements and more complex power models, differences across nodes could also be incorporated into my TCO computation. With the capacity-based approach of distributing applications across the whole HPC cluster (Sect. 4.2.2), the application's percentage p_i of the cluster's capacity (e.g., in node-hours) must also be incorporated into the energy costs, so that:

$$C_{EG}^{pa,n} = \frac{1}{n} \sum_{i=1}^m \left(p_i \cdot n_i \cdot C_{EG,app,i}^{pa,n} \right) \quad (4.17)$$

with the only purpose of $1/n$ bringing this job-mix view into line with the TCO model's general annual cost multiplication by n .

4.4. Discussion of Parameters

After the definition of my productivity model, I discuss in-depth its parameters in terms of range of values, typical examples and use cases, as well as raised challenges to prove its validity and show possible limitations. Especially, the model's

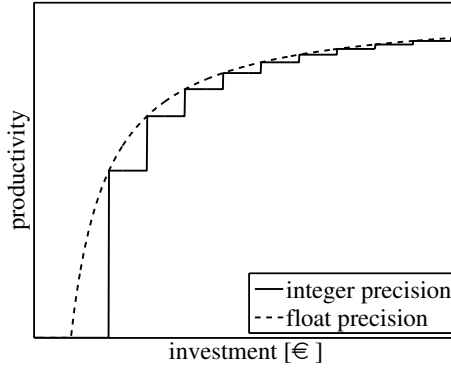


Figure 4.3.: Comparison of productivity as function of investment with number nodes computed as integer values and real values.

predictive power is examined by giving examples on how each parameter might be predicted. An overview of all parameters and their predictability capabilities is given in Tab. A.5.

4.4.1. Number of Nodes n

The number of compute nodes n is a main input parameter of $\Psi(n, \tau)$ and defined by $n \in \mathbb{N}$. Practically, its value may be derived from a given budget I and solving Equ. (4.14). However, this will most certainly yield $n \in \mathbb{R}_{>0}$. While a discretization of results for n must be taken in reality, visualization and interpretation might be more intuitive using continuous values due to smoother results. An example of the differences can be seen in Fig. 4.3. Typical values for the number of compute nodes in a mid-sized HPC cluster as the one at RWTH Aachen University are in the order of 500 to 2,000. A top 5 supercomputer (11/2016) like *Tianhe-2* or *Sequoia* comprises currently in the order of 10,000 to 100,000 nodes.

Node Unit Taking *nodes* as unit instead of cores entails some threats but also opportunities as indicated in the discussion of node-hour units in Sect. 4. Cores are actually the more popular metric as it is used for accounting information and also a metric for the Top500 list. However, especially in pursuit of exaflop computing, core designs get more diverse. For example, CPU cores differ from GPU cores in design and frequency and also from Sunway cores with its MPEs (Management Processing Elements) and CPEs (Computing Processing Elements) used for the #1 Top500 system *TaihuLight* (11/2016). While these differences are still inherit when considering the whole node instead of single cores, the node unit provides an abstraction level that eases the comparison and deemphasizes the core variations. Furthermore, the node unit is also a core parameter for scaling performance tests and for discussions with vendors about prices.

Predictability The number of nodes to purchase is often determined by other constraints such as floor space, cooling or energy infrastructure, or available budget. The latter is probably the most prominent restriction since, usually, a fixed investment is given. Nevertheless, all these factors can be rather well predicted by talking to vendors to get the respective requirements per node (size, power consumption, etc.) or rough ideas on prices per compute node. For example, being able to model the productivity in variation of the number of nodes (or respectively the given investment) is useful to make informed decisions on how to spent the available budget. In Wienke et al. [WIaMM15], we illustrate that for a given use case and showed that investing money into tuning the applications is more productive than using the same money to buy additional hardware.

4.4.2. System Lifetime τ

The system's lifetime τ is the second main input parameter of $\Psi(n, \tau)$ that is usually given in years and defined by $\tau \in \mathbb{R}_{>0}$. Typical values of τ are around 5 years and relate to the vendor's contractual hardware maintenance period. During this time period, vendors will replace broken hardware at no additional cost. After this period, the HPC manager might decide to continue to employ the system. However, since the probability of broken hardware increases with its age and wear, this hardware will often not be replaced and just taken off the cluster. Another impact factor on system lifetime in German university centers is the interval of funding grants. In the past, roughly and simplified, every six years Tier-2 systems get funding. Thus, a system needs to run at least until a new funded system has been installed to verify that science based on HPC can still be carried out.

Variation over Time My productivity model assumes annual dependency of its components, meaning that costs arise equally-distributed across years without further variation. If costs significantly change over years or within the course of one year, my simplified lifetime model must be replaced with more complex expressions, e.g., the integration of components' values over time. Murphy et al. [MNV06] introduce this concept in analytical form for their productivity approach.

Multi-Phase Acquisition With my productivity model, it is possible to compare procurement of hardware in a single phase to multi-phase acquisitions, and even to find the point in time when to invest in novel hardware. The multi-phase setup is motivated by novel technologies promising higher performance, bandwidth and energy efficiency. In [WIaMM15], we illustrated the productivity evaluation of a single-phase and two-phase installation with the objective of having the highest productivity after a certain funding period. Figure 2.2b shows a further example.

Cluster Commissioning Although my productivity model considers the system lifetime τ and the effort and costs for installing the productive software environment, it does not model the time needed for initial hardware installation,

commissioning and acceptance tests that is required before production. If needed, future extensions of the productivity model could include milestones like electrical safety testing, connection of cooling system to facility, integration of cooling system, testing of active components, electrical and cooling testing at load and testing of network connections [JHT16]. Furthermore, a site integration test that ensures that the hardware is correctly integrated into the center’s environment and a user trial where users actually test the system in a pre-production phase over typically a month [JHT16] are part of a procurement process and might be considered in the TCO for a productivity index. Interestingly, observations show that commissioning of big compute clusters takes relatively longer than the one of mid-sized or small-sized clusters, probably due more complex network infrastructures or longer testing phases.

Development Duration If considering further time factors, the likely overlap of productively running the system over τ years with the development time required to port or tune the application must be taken into account. So far, my productivity model assumes that the porting can be done ahead of the system commissioning. However, since the targeted hardware architecture might not be accessible for development tests at this or other sites, the porting and tuning might only start when the system goes into production. Thus, if development effort for this system architecture is high, it will decrease the time that the application can productively run on the system. Future extension of my model will cover this relationship.

Predictability Recapping that the system’s lifetime is likely to depend on the contractual vendor’s maintenance period or the site’s funding period, values for τ can be well estimated and are typically in the range of 4 to 6 years. For longer system exploits, the breakdown rate should be also considered.

4.4.3. Number of Application Runs r

The value of an HPC center and output of my productivity index is defined as the sum over all applications runs with $r(n, \tau) \in \mathbb{N}_0$. This output metric presents a ‘quantification of science’ and has shown real-world applicability in previous works [WaMM13][WlaMM15]. I briefly discuss how it compares to the three main other approaches described in Sect. 3 carried out by (a) performance characteristics (Flop/s, bandwidth), (b) publication counts and funding, and (c) utility.

Performance Metrics Performance characteristics like Flop/s or GB/s have been taken as value of an HPC center either from benchmarks like HPL or from own applications (see my introduction of benchmarks in Sect. 3.5.1). Investigations of own applications need efficiency predictions and a common performance metric across all applications. Thus, the approach by Sterling (Sect. 3.3.2), Kepner

(Sect. 3.3.3) and similar might be valid for single-application perspectives where an appropriate performance characteristic can be found. However, applying this metric to all applications running on a cluster will not appropriately capture the different types of applications or may neglect tuning activities since they are not representable in the selected metric. Instead, the number of application runs r depends on the applications' runtime which is an embracing metric that is valid and relevant for all types of applications. Other works have not shown how to adapt their approaches to such a job-mix setup, but only stated that averages could be taken for multi-application environments without further sensitivity analysis. In contrast, my analysis in Sect. 4.6 indicates that productivity indices differ depending on the chosen application.

Research Competitiveness The value of an HPC center might be expressed in a publication count, the gained amount of external funding (compare Sect. 3.5.2), the number of patents, or the number of nobel prizes. These metrics are difficult to collect in terms of recognizing whether they have been evoked from HPC software runs and, furthermore, they focus on success only. One challenge of success stories is that they might be published or accepted late so that no prediction is possible. For example, the discovery of the Higgs boson in 1964 has only led to a Nobel prize in 2013. In addition, HPC centers also have to support (and pay) for the fundamental or risky research that may need numerous attempts to lead to success or might never reach it at all. Therefore, a scientifically reviewed research grant process accompanies the application process of compute time on an HPC system. There, researchers have to estimate their experimental runs and will include all parts that drive their research forward (even if a results means an exclusion of settings for further research). Hence, my output metric given as the number of applications runs is a valid metric to capture these setups.

Utility The value of an HPC center has been defined in terms of utilities (see Sect. 3.3.1). While a deadline-driven utility might occur for paper deadlines and single realtime-constrained applications, most researchers that run programs on an university HPC cluster will not encounter value loss in certain point in time if they have planned their simulations ahead. Thus, my productivity model does not incorporate deadline-driven utility, but is flexible enough that it could be added if necessary. Preference-driven utilities might be applied in terms of weighting the importance of applications with respect to others. While one might argue that the simulation of a car crash gives more value than the simulation of the manufacturing of a plastic bottle, the question rises who is able to give preferences and estimate weighting factors across different fields of research. A user-based evaluation of their own work with respect to others will probably yield that the own research is the most important one, in contrast, a site-based assessment lacks likely scientific knowledge and power of judgment across all fields and will not be politically reasonable. Hence, I do not incorporate additional weighting factors across applications types into my productivity model.

On the other hand, relying a productivity figure of merit on the center's own application mix (as I do) hinders the comparison across different HPC centers. However, this is a general challenge of productivity indices (presented in Sect. 2.1) and cannot be easily solved.

Predictability The prediction of the number of application runs should be conducted separately by parameter. Therefore, I briefly discuss the predictability of the different factors in the following. An overview is also given in Tab. A.5.

System Availability α

The system availability is defined as $\alpha \in \mathbb{R}[0, 1]$ and typical values comprise 80 % to 95 %. A first estimation approach can be based on previously-experienced system availabilities on that site that will reflect the center's typical maintenance periods or average downtimes for security updates. Vendors might be further able to provide estimates on MTTF, reliability, availability and serviceability (RAS) considerations [MNV06].

Number of Relevant Applications m

Procuring a new HPC system, HPC managers need to consider whether it is an appropriate architecture for the (relevant) applications needing the computational power. For that, starting with the applications that run on the previous cluster is a reasonable approach. While new HPC applications might emerge over time, it would be fortune-telling to base any procurement processes on these. If focusing on the top 50 % or 75 % projects from a capacity perspective, the corresponding number of applications $m \in \mathbb{N}$ can be easily computed from accounting statistics. For example, for the RWTH cluster, 15 projects used 50 % of the available core hours in 2015. Alternatively, HPC managers just define the number of relevant applications m by taking the job mix for the procurement process as foundation or by relying on the projects of the joint applicants of the procurement proposal.

Capacity-Based Weighting Factor p

From the identification process of relevant applications, statistics are available on the node-hour usage and capacity-based percentage that each project run on the previous cluster. Assuming that the amount of research does not change for a project because a novel HPC system has been commissioned, the project's requested node-hour consumption can be directly expressed as percentage p from the potential capacity of the new HPC cluster with $p \in \mathbb{R}[0, 1]$. Nevertheless, the

requested node-hours should incorporate possible performance gains coming with or tuned for the new hardware. The same holds for the less probable case in which the projects' capacity share is assumed to be fixed, i.e., assuming that all projects will 'equally' use more compute time than previously. Here, the percentage factor must first be adapted by accounting for the estimated performance gains, e.g., speedup of factor 2 means the division of the percentage capacity into half, and then scaling it proportionally to the capacity of the new HPC system. Either way, the factor p can be reasonably predicted for the productivity model.

Runtime t_{app}

The model of the application runtime behavior $t_{app}(n) \in \mathbb{R}_{>0}$ is usually provided by the user or HPC expert working with the user. Performance models can be set up to estimate the performance scaling across numerous compute nodes. Linear scaling or Amdahl's Law would be the simplest approaches here. With a performance model in place, the runtime behavior is usually well-understood in terms of compute-bound and memory-bound parts. Giving the details of a new hardware, an estimation of performance behavior can be modeled. For example, an application that comprises a compute-bound part which heavily relies on vectorization might get a speedup of factor two within that parallel part when doubling the vector width. A similar example is shown in Sect. 15.2. Besides hardware characteristics, the application runtime also relates to the amount of development effort spent for parallelization and tuning as modeled by, e.g., the 80-20 rule. This relationship is further discussed in Sect. 7.2 and Part C.

The number of nodes n_i or n_{scale} that an application will efficiently scale to is also provided by the user. It can be based on previous efficiency tests or also assessed using a performance model.

Quality Weighting Factor q_{app}

The quality weighting factor $q_{app} \in \mathbb{R}_{\geq 0}$ accounts for benefits of large-scale runs that are not inherent in runtime improvements but due to increased resolution and bigger data sets, e.g., as with weak scaling runs. This is a factor provided by the user that cannot be concretely defined since it is subjective. Thus, setting it up across all relevant applications is challenging. As rule of thumb, using a factor of one can be the default while $q_{app} > 1$ gives credit to bigger data sets.

4.4.4. Total Ownership Costs

The total costs of ownership cover all expenses that an HPC center and the affiliated researchers have. While I incorporate important TCO factors in my model,

further components could be added. For example, Murphy et al. [MNV06] suggest to include costs for waiting time on job scheduling. Although this waiting time reduces user productivity in terms of continuous investigations, this time will be most certainly used for other activities and, thus, should not directly account for increased costs of HPC centers. This is especially true if the waiting time is in a reasonable range of tens of hours which can be ensured by a sufficient funding structure and reasonable funding periods for new hardware extensions. Similarly, costs for pre-work of the procurement proposal and for the effort to write project proposals for application of compute time could be included. An evaluation of these components is part of my definition of development effort in Sect. 8.1.

I discuss the challenges to quantify and estimate TCO values in detail in Sect. 7.2 of Part B. A summary can be found in Tab. A.5.

4.5. Derived Metrics

The productivity metric introduced can be applied in numerous scenarios. However, in certain occasions a slight extension of the productivity metric or a derivation of it can be more appropriate. Here, I demonstrate some alternatives with corresponding use cases.

User Productivity While I have focused on the computing center’s perspective on productivity so far, an alternative way could run productivity from a user’s perspective. For example, including all costs and interpreting productivity as function of the development time, the developer can estimate how much tuning effort should be invested (see Sect. 2.3). If reducing the cost to development effort only, the productivity index approximates approaches shown in Sect. 3.2. Especially, if the reduced productivity metric of setup A is taken in comparison to setup B, then my productivity can be put in line to the the relative development time productivity (Sect. 3.2.1).

Costs Per Application Run A very simple derivation of the productivity metric is given by its reciprocal that defines costs per application run. In a multiple-job HPC landscape, this comes down to an *average* cost per application run (across different applications). In a single-job perspective, this metric may provide cluster users a tangible indication on the cost of their science. Users usually have an easy access to this price/performance ratio since it is well-known from everyday life, e.g., 1.09€ per 100 g chocolate. On the other hand, this reciprocal might produce very small values that complicates its interpretation. We showed the application of this metric in [WaMM13].

Cash Flows & Present Values Taking up the economic definition of productivity as cost effectiveness, accounting for discounted cash flows (DCF) is common in economics. DCF analysis is based on the *opportunity cost* of money, i.e., money is worth more today than tomorrow (even without inflation) [Tas03, pp. 24]. To account for that, a discount rate d (e.g., 3%) is used to convert all monetary values C in the cost-effectiveness ratio to their *present values* (PVs) [CK10, p. 518]: $PV = \sum_{t=1}^{\tau} \frac{C_t}{(1+d)^{t-1}}$ with t the compounding period (here one year). The NPV further extends this approach by incorporating cash inflows and outflows. While outflows are covered by the present values of the total ownership costs of an HPC system, inflows represent the present values of the HPC system’s benefits, so that $NPV = PV(\text{benefits}) - PV(\text{costs})$.

The output of decoupling the present value calculations from this cash-flow perspective closely relates to the BCR definition given in Sect. 2.1. However, since my productivity index does not define the benefit of an HPC system in monetary units, but as number of application runs on the HPC system over its lifetime, an adaption of the perspective on cash flows could be taken by focusing on the (cost) denominator of the productivity metric only. For instance, (waste) heat that is produced by powering the cluster can be re-used for heating buildings as seen at the Leibniz Supercomputing Centre [Lei17]. Corresponding heating savings can be seen as inflows in an adapted cash-flow perspective.

Return On Investment The return on investment is also a cash flow metric and is usually presented as ratio of net investments gains (or losses) divided by total investments. Thus, in contrast to my productivity model, it takes up the benefit-cost ratio (instead of the cost-effectiveness approach) and further covers net income or profits as numerator. In financial markets, this percentage perspective is popular since it also allows to compare businesses of different sizes. In HPC, ROI has been used in different flavors to argue for investments in HPC fields, e.g., as incentive or justification to governmental entities (see Sect. 3.4.2). For example, IDC used three HPC ROI models based on (a) revenues/ GDP generated (b) cost-savings/ profits generated, and (c) jobs created [JCD13].

Break-Even Analysis In cost-benefit analysis, the break-even point describes the point at which $BCR = 1$, i.e., benefits and costs are equal. For my productivity index and its derivations, break-even analysis can also be applied to compare different setups and determine *when* one setup gets beneficial over another. It may replace single direct comparisons of specific scenarios that are conducted to choose the best option from that pool. For example, break-even investment as illustrated in Fig. 2.2a describes the amount of money from which on the productivity of system B is continuously higher than the one of system A. Another use case is break-even effort: If the procurement of a novel architecture type entails considerable

porting effort, the break-even effort would indicate the maximum amount of porting effort allowed so that the new system still pays off. Examples for break-even investments and break-even efforts can be found in our publication [WaMM13] that compared the worth of accelerators over an x86-based architecture. Corresponding formulas are published in the TCO spreadsheet on our webpage [Wie16a].

4.6. Uncertainty and Sensitivity Analysis

Every model of a (physical) phenomena is prone to variances in its output due to several types of errors that may be introduced into the input data (parameters). This also holds for the prognostic productivity model whose estimated value is based on numerous assumptions in its parameters as discussed in Sect. 4.4. Especially, these (potentially inaccurate) assumptions and values that base on experimental measurements, e.g., application runtimes, power consumption or development efforts, contribute to the uncertainty of the model output. To scrutinize the model by quantifying the variance of the model output and detecting the important parameters that impact this output variance, uncertainty and sensitivity analysis are applied, respectively.

Uncertainty and sensitivity analysis can be easily combined with uncertainty analysis preceding sensitivity analysis [SRA⁺08, p. 1]. In Sect. 4.6.1, uncertainty analysis of the productivity model is carried out by focusing on defining output variance as probability density function gained by simulation-based sampling. It is followed by a global sensitivity analysis of the productivity model taking a variance-based approach (VBA) in Sect. 4.6.2. In both sections, basic information on the different techniques comes from Loucks et al. [LVBS⁺05], while details on uncertainty analysis are taken from the *guide to the expression of uncertainty in measurement (GUM)* [BIP08] and details on global sensitivity analysis from Saltelli et al. [SRA⁺08].

4.6.1. Uncertainty

The GUM [BIP08] defines uncertainty of measurement as “parameter, associated with the result of a measurement, that characterizes the dispersion of the values that could reasonably be attributed to the measurand”. In productivity modeling, the required measurand is represented by the productivity index of whom some functional parameters can also be viewed as measurands such as the software development effort estimator. The measurand’s uncertainty yields from errors in the input parameters, i.e. random and systematic errors (bias). While an error is defined as the “difference between the measured value and the *true value* of the thing being measured” [BIP08] and unknown, uncertainty is a “quantification of

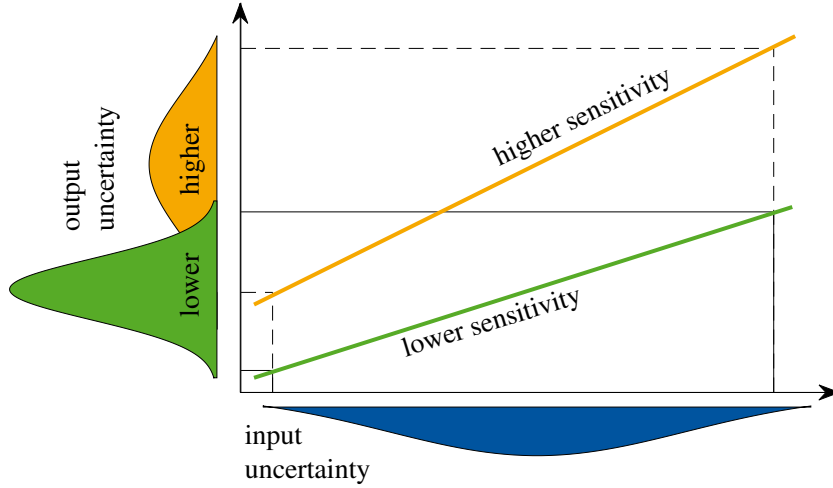


Figure 4.4.: Schematic relationship of uncertainty in a linear model based on PDFs (similar to [LVBS⁺05][LOVZ97]): The errors in the input parameter with low sensitivity yield low output uncertainty (green), whereas the errors in the input parameter with high sensitivity cause higher output uncertainty (orange).

the doubt about the measurement result” [BIP08]. Uncertainty is determined by propagating the uncertainty of input components through the model (uncertainty propagation, often also called error propagation). For that, input components are attributed with probability distributions that are either derived from an observed frequency distribution (so called Type A uncertainty) or from an assumed (subjective) probability density function (so called Type B uncertainty) [BIP08], e.g., taken from manufacturer’s specification or common sense. The uncertainty of the productivity model is mainly based on Type B evaluations of the input parameters since most of these values come from assumptions based on past experience, manufacturer’s specification or calculations, e.g., the system’s lifetime, hardware price or energy specification and performance models of application runtime.

Uncertainty analysis has been subject of much research [LC08]. Here, the GUM provides the theoretical approach based on algebraic definitions. In contrast, the methods of simulation-based uncertainty propagation [LC08], e.g., using Monte Carlo sampling, may be seen as “experimental statistics” [FF14] that got common practice for problems where the pure algebraic approach would be difficult. For example, simulation-based approaches can be easily applied to non-linear models and further provide a graphical representation of the probability density function (PDF) of the measurand. Figure 4.4 illustrates schematically the relationship of input and output probability distributions (in the simplified linear case). Depending on the shape of the output PDF, model uncertainty is referred to as low or high. However, whether a gained uncertainty is acceptable for a certain model must be manually defined. In addition to an available PDF representation, uncertainty of the model output is usually quantified using the sample mean $\bar{x}$ (see

Equ. (4.18)), median, sample deviation σ , variance V (refer to Equ. (4.19)), quantiles and confidence bounds [SRA⁺08, p. 7]. The (potential) shape of a PDF can further be determined by its skewness s (given in Equ. (4.20)) and kurtosis k as defined in Equ. (4.21).

$$\bar{x} = \mu = \frac{1}{n} \sum_{i=1}^n x_i \quad (4.18)$$

$$V = \sigma^2 = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2 \quad (4.19)$$

$$s = \frac{E(x - \mu)^3}{\sigma^3} = \frac{\frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^3}{\left(\sqrt{\frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2} \right)^3} \quad (4.20)$$

$$k = \frac{E(x - \mu)^4}{\sigma^3} = \frac{\frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^4}{\left(\frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2 \right)^2} \quad (4.21)$$

For the uncertainty analysis of the productivity model, I apply a simulation-based approach of uncertainty propagation due to the non-linear character of the productivity model. Furthermore, uncertainty analysis is part of the global sensitivity analysis (conducted in Sec. 4.6.2) that is also based on simulation-based sampling. Thus, computational evaluations of the productivity model can be re-used. Results for several use cases are presented in Sect. 4.6.3.

4.6.2. Sensitivity

Sensitivity analysis (SA) is defined as “study of how uncertainty in the output of a model (numerical or otherwise) can be apportioned to different sources of uncertainty in the model input” [STCR04]. Sensitivity analysis can help improving and justifying a model’s quality by revealing technical errors in the model, attributing model uncertainty to specific input parameters, using these results to increase the model precision or simplify the model, or opt for certain policy assessments. Here, SA follows the objective to identify the important factors where *factor* represents an uncertain model input. However, insufficient definitions of importance often lead to ambiguous results [SRA⁺08]. Therefore, Saltelli et al. [STCR04, pp. 52, pp. 109] define *settings* that map definitions of importance to well-identified sensitivity measures. The two relevant settings for this study are:

- Factor Priorization (FPR): Detect factors that create most of the output variance. Consequently, estimation or measurement of these factors should optimally be improved.
- Factor Fixing (FFI): Identify factors that do not have a significant contribution to the output variance. Thus, these can be fixed within their range of variation and thereby also simplify the model.

Another challenge is to ‘correctly’ define the model inputs. Input parameters may actually seem fixed, but could have a great effect on the model output if varied. On the other hand, putting all parameters as input variables into the sensitivity analysis might allow the model output to vary widely preventing any practical use. Therefore, a trade-off should be considered that Leamer [Lea83] defines as “global sensitivity analysis in which a neighborhood of alternative assumptions is selected and the corresponding interval of inferences is identified. Conclusions are judged to be sturdy only if the neighborhood of assumptions is wide enough to be credible and the corresponding interval of inferences is narrow enough to be useful.” Practically, a few key inputs create the majority of uncertainty so that adding more model inputs may not increase the model output.

Two ways to address SA are local and global sensitivity analysis [PBF⁺16]. Local sensitivity analysis captures output variation due to input variation (locally) around a specific (single) point. They are based on derivatives and can often be found in literature [SRA⁺08, p. 11]. Instead, global sensitivity analysis (GSA) explores the entire space of input variation. Therefore, the risk of overlooking important factors is lower than with local methods. To evaluate the sensitivity of the productivity model (factor prioritization and factor fixing), I use the integral GSA approach by applying variance-based sensitivity analysis (VBSA). I argue this choice over local SA and explain its definition in the following.

Any model can be represented in the form $Y = f(X)$ where X is a vector of k model inputs $X = (X_1, X_2, \dots, X_k)$. We assume that all X_i are independent and have a nonnull range of variation. The derivative-based SA approach uses the partial derivatives as sensitivity measures:

$$S_{X_i}^p = \frac{\partial Y}{\partial X_i}. \quad (4.22)$$

Obviously, they can only be evaluated at a base point, i.e., a nominal value given by the modeler. An extension of the measure $S_{X_i}^p$ is the sigma-normalized derivative:

$$S_{X_i}^\sigma = \frac{\sigma_{X_i} \partial Y}{\sigma_Y \partial X_i} \quad (4.23)$$

where σ_{X_i} is the input standard deviation and σ_Y the output standard deviation. It improves on the meaningfulness of the ranking by relativizing the order and normalizing it to one. For linear models, the sigma-normalized derivative coincides with the standardized regression coefficient $\beta_{X_i} = S_{X_i}^\sigma$. This regression-based method assumes uncertainty propagation from Monte Carlo experiments and further (sigma-)normalizes regression coefficients gained from least-square computation:

$$\beta_{X_i} = \frac{\sigma_{X_i}}{\sigma_Y} b_{X_i} \quad (4.24)$$

with b_{X_i} from linear regression of form $y(i) = b_0 + \sum_{j=1}^k b_{X_j} x_j^{(i)}$. While derivative-based approaches only work well for pure linear models for which the rest of the space can be explored by linear extrapolation, the regression-based method is also reliable for non-linear models with a high degree of linearity. Instead of using a single base point, the β 's are averaged among multiple dimensions and explore the entire space of the model inputs. Note that both formulas (4.23) and (4.24) decompose the variance of the model output. In the following, I accept output *variance* as proxy for uncertainty as most practitioners do [SRA⁺08, p. 20]. Variance may be mainly an inappropriate uncertainty measure if the output distribution is multi-modal or highly skewed [PBF⁺16].

Since my productivity model introduced in Sec. 4.1 has a non-linear character, I focus on a *model-free* approach that is based on averaged partial variances. For that, the output uncertainty is investigated under the condition that factor X is fixed at a particular value x_i^* . The corresponding conditional variance of Y is written as $V_{X_{\sim i}}(Y|X_i = x_i^*)$ taken over $X_{\sim i}$, i.e., all factors but X_i . To remove the dependency to point x_i^* and eliminate rare combinations that may cause extremely high variances, the average of this measure over all possible points x_i^* is taken: $E_{X_i}(V_{X_{\sim i}}(Y|X_i))$. Looking at it from the opposite side, first, the expected values over X_i being fixed to the points x_i^* can be computed and, then, the variance is taken: $V_{X_i}(E_{X_{\sim i}}(Y|X_i))$. In fact, it holds [SRA⁺08, p. 21]:

$$E_{X_i}(V_{X_{\sim i}}(Y|X_i)) + V_{X_i}(E_{X_{\sim i}}(Y|X_i)) = V(Y). \quad (4.25)$$

Thus, a small $E_{X_i}(V_{X_{\sim i}}(Y|X_i))$ or a large $V_{X_i}(E_{X_{\sim i}}(Y|X_i))$ means that X_i is an important factor. The sensitivity index

$$S_i = \frac{V_{X_i}(E_{X_{\sim i}}(Y|X_i))}{V(Y)} \quad (4.26)$$

with $0 \leq S_i \leq 1$ is called the *first-order effect* or *main effect* of X_i on Y . Since higher S_i mean more importance of input X_i , S_i can be used for the FPR setting. Furthermore, S_i equals $\beta_{X_i}^2$ (refer to Equ. (4.24)) for linear models. For additive models, i.e., impacts of model inputs can be separated by variance decomposition, it holds that $\sum_{i=1}^r S_i = 1$. For non-additive models, the first-order effects do not add up to one, so that $\sum_{i=1}^r S_i \leq 1$ holds for the model-free generalization. The reason for this is that first-order terms cannot detect interaction effects among two factors X_i and X_j expressed as S_{ij} . Following this considerations for higher dimensions, the interaction terms of k inputs can be written as [SRA⁺08, p. 162]:

$$\sum_i S_i + \sum_i \sum_{j>i} S_{ij} + \sum_i \sum_{j>i} \sum_{l>j} S_{ijl} + \cdots + S_{123\cdots k} = 1 \quad (4.27)$$

that is based on Sobol's variance decomposition approach. However, in practice, evaluating all $2^k - 1$ terms of Equ. (4.27) is often too compute expensive. Therefore,

the concept of *total effects* is introduced: It “accounts for the total contribution to the output variation due to factor X_i , i.e its first-order effect plus all higher-order effects due to interactions.” [SRA⁺08, p.162]. For instance, the total effect of X_1 having three model inputs is: $S_{T1} = S_1 + S_{12} + S_{13} + S_{123}$. Generally speaking, Equ. (4.25) is re-used and conditioned with respect to all model inputs but X_i :

$$V(Y) = V(E(Y|X_{\sim i})) + E(V(Y|X_{\sim i})) \quad (4.28)$$

where $E(V(Y|X_{\sim i}))$ is the remainder of the output variance that represents the (average) share under the assumption that all $X_{\sim i}$ could be fixed to their true values. Since the true values of $X_{\sim i}$ are unknown, the average is computed over all possible combinations of $X_{\sim i}$. From that, the total effect index of X_i can be derived:

$$S_{T_i} = \frac{E(V(Y|X_{\sim i}))}{V(Y)} = 1 - \frac{V(E(Y|X_{\sim i}))}{V(Y)} \quad (4.29)$$

The total effect indices can be used to determine non-influential model inputs because they will now show any effect if fixing them anywhere over their range of variability. For that $S_{T_i} = 0$ is necessary and sufficient condition (FFI setting).

The main effect S_i and total effect S_{T_i} are good descriptors of the model’s sensitives:

- The measures are model-free, i.e., independent of the linearity, monotonicity, and additivity of the model.
- They have the capability to explore the impact of the full range of variation of each input parameter (in contrast to derivative-based methods).
- They can detect interaction effects among model inputs (in contrast to derivative- and regression-based approaches).
- The application of total effect indices as replacement of all higher-order effect terms reduces the computational cost. For example, for $k = 10$ only 20 instead of 1,023 terms must be evaluated.

Nevertheless, the computational cost is still very high since estimating sensitivity coefficients requires numerous model evaluations. Thus, the acceleration of computing the sensitivity indices is currently a large research field [SRA⁺08, p.38].

I use the approach by Saltelli et al. [SRA⁺08] explained in Appendix A.5. They try to reduce the sample size by approximation of the sensitivity coefficients. For that, they use a numerical procedure based on Monte Carlo experiments for sampling the input parameters of the productivity model within their range of variability given by their PDFs. For the application of Saltelli’s VBSA approach, I take (and extend) the methods from the SAFE Toolbox [PSW15] that supports my Matlab implementation of the productivity model (see Sect. 4.7).

4.6.3. Results

Since a simulation-based uncertainty and sensitivity analysis works with probability distributions, it is always dependent on concrete values, e.g., mean or standard deviation, derived from a particular project. The project also specifies constant parameters and defines application-dependent behavior, e.g., performance models or effort estimation. Thus, results presented here have no claim of being universally valid. Instead, I analyze two real-world case studies that cover different classes of applications in different setups: First, I look into the engineering application *psOpen* that can scale strongly up to 16,000 compute nodes using MPI. A description of the application, of conducted HPC activities and corresponding application-dependent productivity data can be found in Appendix A.1.2. Second, I investigate the bio-medical application *NINA* (see Appendix A.1.1) that runs on a single compute node leveraging CPUs or accelerators with different parallel programming models. Note that these analyses are performed a posteriori, while a case study that applies ex-ante procurement analysis can be found in Sect. 15.2. Here, a posteriori analysis means that application-dependent data such as development effort, runtime and power consumption could and were measured on the targeted hardware architecture. I disturb this data to simulate uncertainty by a-priori estimation. System-dependent parameters are mostly approximated anyway so that they likely introduce errors. The assumptions on the PDFs of the changeable parameters are given in Tab. 4.2. Samples with $N = 30,000$ are generated using Latin Hypercube sampling and Saltelli's VBSA approach.

I start with the evaluation of my productivity model from the single-application perspective introduced in Sections 4.2.1 and 4.3.1. With that, I analyze the sensitivity of small-scale and large-scale setups. Furthermore, I show how different hardware and software environments can be compared giving the uncertainty in productivity results. Finally, the composition of multiple applications to a job mix and how the capacity-based weighting factor affects the productivity result is investigated. This leads to a validity check of my proposed productivity metric with respect to the assumption that only m relevant applications are modeled.

Single-Application Perspective

Results from the single-application perspective are based on system details that we have quantified in the context of the RWTH Compute Cluster and described in detail in Wienke et al. [WaMM13][WiMM15]. I explain these assumptions and quantifications in Appendix A.2.2. Table A.6 gives an overview of all productivity parameter values and determines the parameters varied for the VBSA.

Table 4.2.: Assumptions for PDFs of productivity parameters.

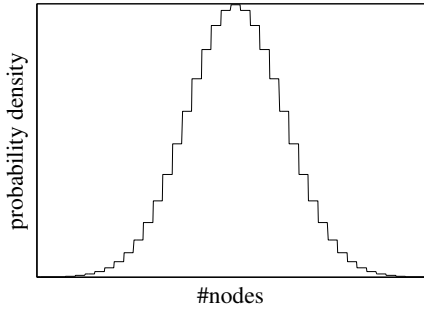
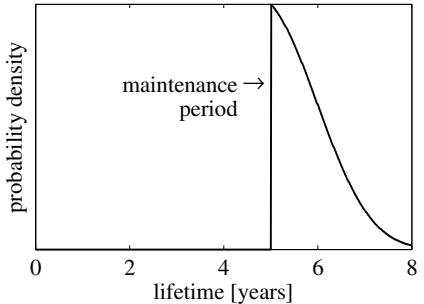
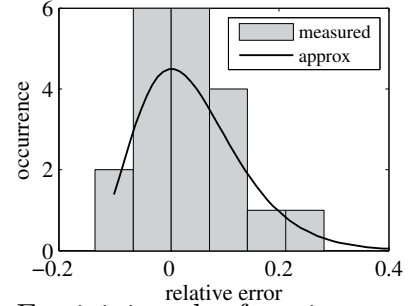
Parameter	PDF	Remarks/ assumptions
#nodes n, n_{scale}	discretized (positive) normal distribution with μ estimated and $\sigma = 0.1 \cdot \mu$, discretization/ approximation by binomial distribution with $p = 1 - \frac{\sigma^2}{\mu}$, $n = \frac{\mu}{p}$	budget I is fixed, from that n is derived (see Sect. 4.4.1)
		
system lifetime τ	modified Weibull distribution with $a = 5$, $b = 0.5$: probability of zero for values lower than maintenance period ($t = 5$ years)	lifetime distribution of electronic components (Weibull), components get replaced within maintenance period (see Sect. 4.4.2)
		
effort DE : cone of uncertainty	uniform distribution in the range of $[0.25 \cdot DE_\mu; 4 \cdot DE_\mu]$ with DE_μ the most effort probable value	the cone of uncertainty (Sect. 9.2.2) assumes that a factor of 16 between the low and high end of effort estimation exists in the early software project phase in the best case

Table 4.2.: Continued from previous page

Parameter	PDF	Remarks/ assumptions
effort <i>DE</i> : self reports (default)	modified (positive) normal distribution with μ estimated and $\sigma = 0.1 \cdot \mu$, disturbed by errors between real and reported efforts (see Sect. 11.3.1). I assume that $f_X(x)$ and $f_Y(y)$ are stochastic independent: $f_{XY} = f_X(x) \cdot f_Y(y)$ where $f_X(x)$ represents the error distribution and $f_Y(y)$ the normal distribution. For the product of two continuous random variables $Z = X \cdot Y$, it holds $f_{X \bullet Y} = \int_{-\infty}^{\infty} f_{XY} \left(\frac{z}{y}, y \right) \cdot \frac{1}{ y } dy$.	rough effort can be estimated on base of reported data, error distribution between observed and reported errors taken from Perry et al. [PSV95] (also compare Sect 8.1.1). I approximate the corresponding CDF of the discrete error distribution by a gamma CDF with $a = 6.0597$ and $b = 0.0388$:
		 For joining the functions to $f_{X \bullet Y}$, f_X is shifted so that a relative error of zero equals one, i.e., normal distribution f_Y is not modified.
others	positive normal distribution with μ and $\sigma = 0.1 \cdot \mu$, if $x < 0$ set $x = 0$, if $\mu = 0$ set $\sigma = 0.1$ and $x = \text{abs}(x)$	mean is approximated value with some uncertainty

Small-Scale Cluster For a small-scale setup, I investigate the engineering application *psOpen* which is executed on the integrative hosting [ITC16] hardware that an RWTH institute has acquired for roughly 250,000 €. Based on hardware prices used in [WIaMM15], this investment can be mapped to roughly 56 WST compute nodes (see Tab. A.4) that run over 5 years with a system availability of 80 %.

First, I perform an uncertainty and sensitivity analysis for the given setup with the assumption of 10 % standard deviation in normally-distributed parameters.

Figure 4.5a illustrates the propagated uncertainty function of the corresponding productivity with a mean of 27.25 and standard deviation of 6.18. Using a confidence level of 95 %, the corresponding confidence interval ranges from 17.31 to 41.29 (dashed lines). From Fig. 4.5b, it is evident that the variation in productivity is mainly due to uncertainties in system lifetime τ that account to a main effect of 35 %. Furthermore, productivity is sensitive on the center’s system availability, the serial runtime of the application, and the (tuned) kernel runtime. These parameters directly influence performance and, thus, the number of application runs. Since this setup does only contain small manpower, software and maintenance costs, performance variation remains the main impact factor.

Increasing the standard deviation of the normally-distributed parameters to 20 % yields a higher output uncertainty with confidence bounds at 12.04 to 58.05 (dashed lines in Fig. 4.5c). Nevertheless, its mean value is close to the one in Fig. 4.5a. Contrarily, the main effect of the serial runtime increases from 20 % to 32 % with higher variation, while the main effect of the system lifetime decreases respectively (Fig. 4.5d). Most other parameters just slightly increase in effect.

Thirdly, I investigate the impact of the type of assumed probability distribution by changing all previous normally-distributed parameters to uniform PDFs within realistic ranges. Fortunately, the change in PDF does not yield a higher uncertainty in productivity with a confidence interval from 11.72 to 29.93 and as evident in Fig. 4.5e. Similarly, the main effects of most parameters do not differ greatly with respect to their normally-distributed counterparts (see Fig. 4.5f). Only the effect of the software license and compiler costs increases to 27 % since I now assume uniform variations between 0 € and 50,000 €.

Overall, this small-scale cluster setup indicates that few (performance-related) parameters must be estimated precisely to achieve little uncertainty in productivity. Since these parameters are well understood and therefore reasonably predictable, in addition to a diminishing impact of the kind of underlying PDF, I conclude that my productivity model is robust within the given conditions.

Large-Scale Cluster For a larger-scaled example, I set the number of nodes to 500. I also assume that more development effort is needed to tune the application for this computing center scale. Thus, I set the basic estimated value to one year (=210 days) of development, as well as 10 days of annual maintenance effort. The remaining parameters are setup in accordance to the first small-scale example (using 10 % of mean for all normally-distributed parameters).

When running the uncertainty analysis, the productivity PDF of *psOpen* delivers a mean of 24.02 with standard deviation of 5.36. The mean is slightly lower than the one from the small-scale example due to the increased development and maintenance effort. Nevertheless, the (higher) variances in these two parameters

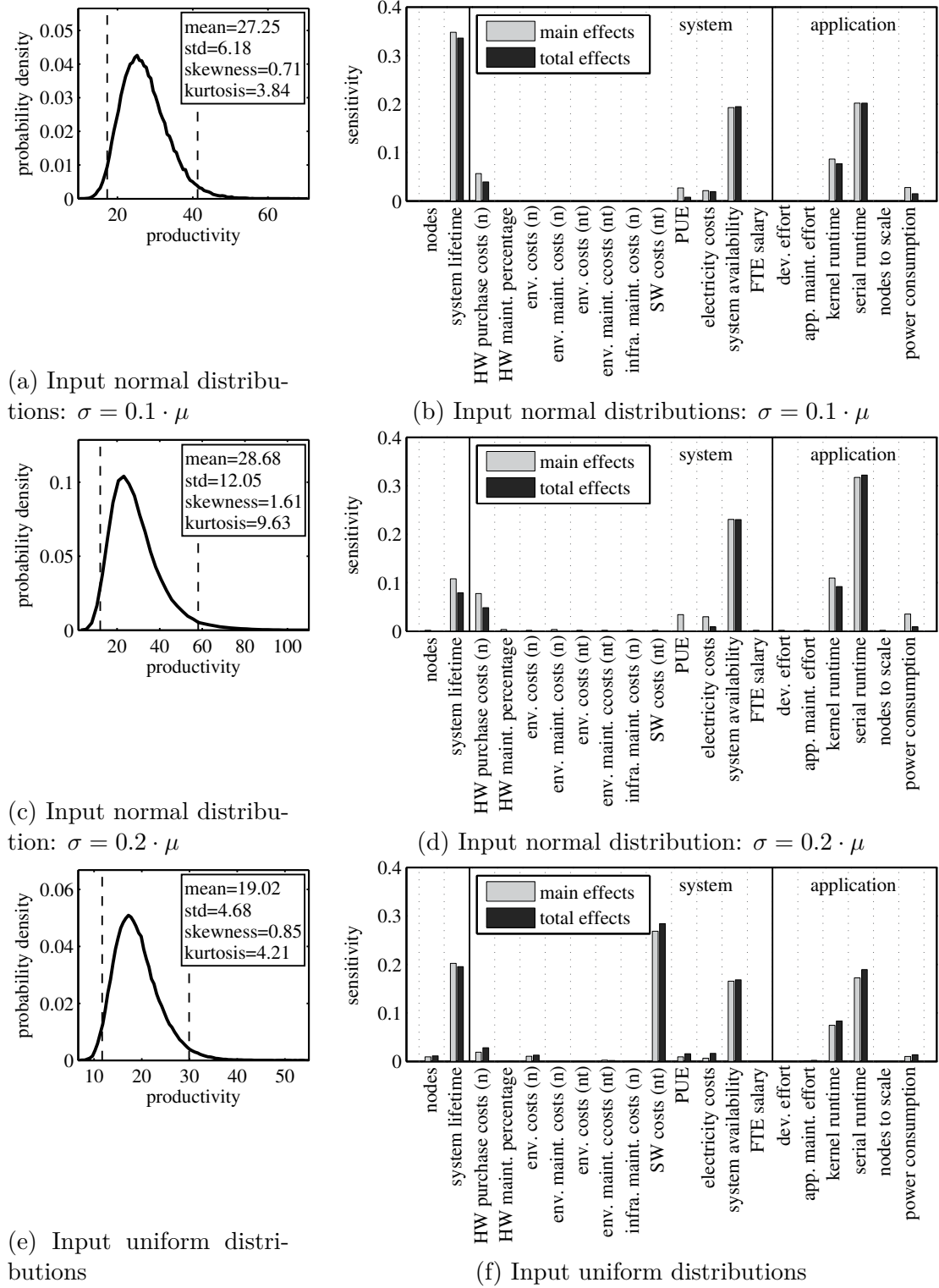


Figure 4.5.: Uncertainty and sensitivity analysis of *psOpen* for the small-scale setup. Left: output uncertainty as PDF, right: main and total effects.

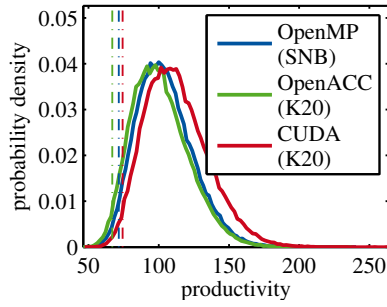


Figure 4.6.: PDFs of *NINA* with different HPC setups.

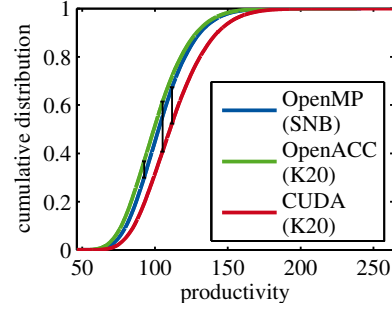


Figure 4.7.: CDFs of *NINA* with different HPC setups.

still do not contribute directly to the variance in productivity which is evident in the corresponding low main effects. Further, in comparison to the small-scale setup, the sensitivity analysis shows a similar behavior in effects with respect to the remaining parameters.

In another sensitivity analysis, I investigate the impact of the *cone of uncertainty* (as introduced in Sect. 9.2.2) that suggests a factor of 0.25 to 4 as uncertainty in traditional software development (without focus on HPC) in the best case. Distributing effort values uniformly within this range, i.e., $[0.25 \cdot 210; 4 \cdot 210]$, the resulting main effects remain still below 1%. Only with a high factor range of 1/16 to 16, the main effect of development effort reaches 5%.

Hence, the productivity metric is not very sensitive towards errors in development effort and it seems that a precise estimation of development effort is not important. However, especially, when buying novel computer architectures, an overlap of software development with the system lifetime is inevitable, i.e., valuable system lifetime may be decreased correspondingly. In this case, a well-estimated development effort also reduces the uncertainty in system lifetime and its propagation to productivity. Additionally, when moving from the HPC center's perspective to the user's perspective, the precise estimation of software development effort remains an important component in project management and workforce allocation.

Comparison of HPC Setups Uncertainty in productivity values complicate the direct comparison of HPC setups due to possible overlap in their PDFs. In this case, to check for differences in productivity of various HPC setups, a significance test must be applied. The Kolmogorov-Smirnov statistic tests whether two samples come from the same population, especially it tests for differences in variance, skewness and kurtosis [HS16, pp. 550] by using the maximum vertical deviation between the two corresponding empirical cumulative distribution functions (CDFs).

To illustrate this comparison of HPC setups, I look at the real-world code *NINA* that has been optimized with OpenMP for a two-socket Intel Sandy Bridge (SNB

in Tab. A.4) server and that has been ported with OpenACC and with CUDA to an NVIDIA Kepler GPU (K20 in Tab. A.4). The center's setup is similar to the one previously introduced where I adapted purchasing costs and maximum energy consumptions to the corresponding hardware architectures and increased the hardware maintenance percentage to 8.2 % with respect to the corresponding contracts. The application-dependent characteristics can be found in Appendix A.1.1. As with the small-scale cluster example, I assume that the investment is 250,000 € from which I compute the initial number of compute nodes that can be most probably acquired. Thus, the comparison of HPC setups may base on different numbers of compute nodes for each setup.

When applying the productivity model without any uncertainty disturbances, the productivity metric delivers a single value for each setup that can be clearly ranked. In Fig. 4.6, these values are represented as dashed lines and lie close together: OpenACC on the GPU with 66.97, OpenMP on the CPU with 71.61, and CUDA on the GPU with 74.35. When including uncertainties in the parameters similar to the small-scale cluster setup, the single productivity values are either slightly below or above the lower bound of the confidence interval. This orientation to the left is mainly due to the assumed PDF of the system lifetime. Furthermore, the three productivity PDFs of the different setups show a huge overlap on first sight as given in Fig. 4.6. However, their CDFs (compare Fig. 4.7) indicate some differences that are further tested with the Kolmogorov-Smirnov statistic. This two-sided test reveals that all three setups differ significantly (illustrated by black markers) on a 5 % significance level. Hence, it is valid (within the significance level) to state that one HPC setup is more productive than the others in the context of the given use case.

Job-Mix Perspective

For the cluster setup where multiple applications are running, I focus on the capacity shares p that are defined for each application (compare Equ. (4.10)). For that, I setup a simple (artificial) job mix that contains two applications. The first application is *psOpen* with the same configuration as in the large-scale cluster example. This configuration only differs in the denoted capacity share that is now set from $p = 1.0$ to $p = 0.75$. The second application accounts for $p = 25\%$ of the cluster's capacity, respectively. The runtime of this second application is modeled by Amdahl's Law with a serial runtime of $t = 1,000$ s, a parallel portion of 80 % and the number of scaling nodes $n_{\text{scale}} = 10$. The sensitivity analysis of this setup shows that parameter variations of the first application have a higher impact on productivity variance than the ones of the second application (Fig. 4.8).

Extending the analysis of the relation from capacity shares to productivity, I examine the validity of the reduction from the full job mix to m relevant applications

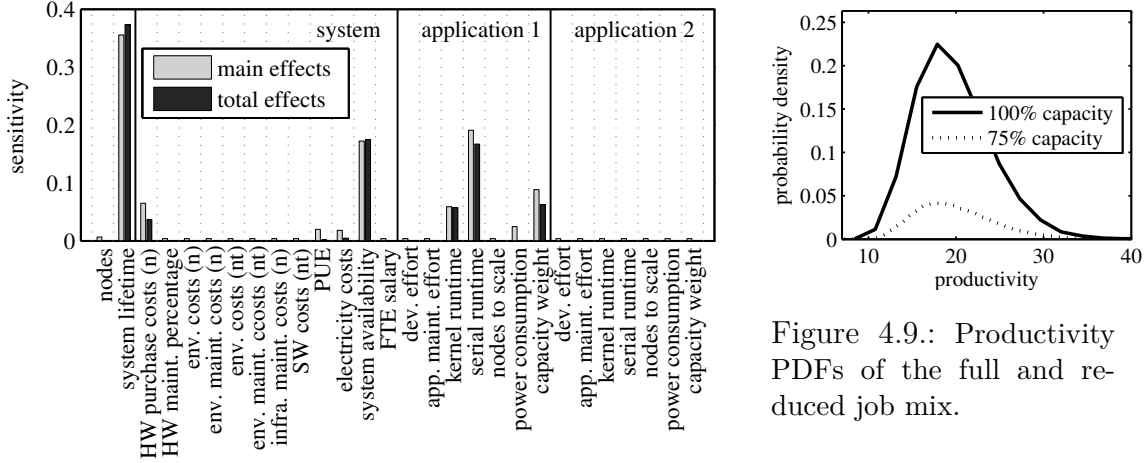


Figure 4.8.: Main and total effects of the full job mix.

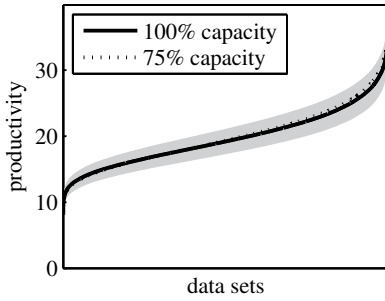
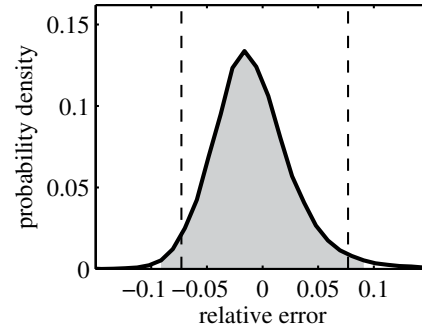
Figure 4.10.: Productivity variation for $N = 30,000$ samples ordered by size: full and reduced job mix.

Figure 4.11.: PDF of the relative error between full and reduced job mix.

as proposed in Sect. 4.2.2. The literature provides numerous approaches to check the validity of an uncertain model towards real (experimental) data. However, it does not reveal a common approach to test the validity of a simplified model, i.e., here the reduction of applications, against the reference model, i.e., the full job mix. The latter approach differs in the way that the simplified model does not only introduce a random error, but an additional error β due to its simplification that the modeler is willing to accept. Therefore, I take the following approach: First, I sample the full job-mix setup with both applications fixed to their capacity shares. The same sample set is applied to my proposed (simplified) model that contains here just the *psOpen* application with $p = 0.75$. The corresponding probability density functions can be found in Fig. 4.9 where the pure line represents the job mix with a total of $p = 1$ and the dashed line the reduced model. Second, I define to accept my reduced model when its productivity values lie within $\beta = 10\%$ deviation from the full job-mix model. Figure 4.10 illustrates the according productivity metrics as function of the given samples where the gray area represents

the 10 % range. To make the acceptance range more evident, I directly compute the relative error for all productivity values of the simplified to the reference model. The PDF of relative errors can be found in Fig. 4.11 where all errors in the range of $[-0.1; 0.1]$ are marked by the gray area. Since ‘most’ values are within this range, I accept the simplification. To formalize the term ‘most’ values, I further compute the confidence bounds on a 95 %-level for these relative errors (dashed lines in Fig. 4.11). Since the 10 %-deviation is completely covered by the confidence interval, my proposed (reduced) productivity model is acceptable for the given setup. It is noteworthy that other setups can yield productivity differences that are not acceptable as they strongly depend on the number of applications runs and costs that are produced by the remaining left-out applications.

Summary

In the context of several use cases, this uncertainty and sensitivity analysis illustrates my model’s robustness against several uncertainty factors. With respect to factor prioritization, I identified only few performance-related well-understood parameters that must be accurately predicated to reduce the variance in productivity. The analysis also shows that resulting productivity metrics can be used to compare different HPC setups even with the given uncertainties in the parameters. Finally, it supports the reduction to a limited number of relevant applications by showing little difference in productivity PDFs.

4.7. Tool Support

Having introduced a mathematical model for productivity in HPC, its practical application for real-world setups should be supported by computational aids. Especially, due to the great amount of parameters, the numerical computation is preferable over the analytical one. We provide three tool alternatives to practically compute productivity.

Spreadsheet First, a spreadsheet can be downloaded from our webpages of the HPC Chair of RWTH Aachen University [Wie16b]. It has been developed with respect to our research presented in [WaMM13]. It covers the basic functionality of productivity computations for single-node application runs that abstract the job-mix cluster view to single-application employment. It computes the productivity’s reciprocal expressed by the *cost per application run* (see Fig. A.1). If comparing several setups, it can further exploit the break-even investment, i.e., the minimum investment needed so that one setup is beneficial over another. The spreadsheet implementation also reduces the possibilities of effort-, performance-

and energy-dependent model propagation to the most basic ones. Although these simplifications seem to constrain the power of the productivity model, they rather promote its application in certain setups: The simplification enables an easy access to first productivity numbers, while it can still be extended with basic spreadsheet knowledge. Therefore, it is especially suited for the manager perspective. Furthermore, this basic approach avoids the evaluation of complex formulas and thereby restricts computational cost.

Aachen HPC Productivity Calculator - aixH(PC)^2 To be able to support a more complex version of the productivity implementation based on [WlaMM15], the tool *aixH(PC)^2 - Aachen HPC Productivity Calculator* is provided online. It has been implemented from the model template with Java Script by our apprentice Jonas Hahnfeld and can be easily used via the web interface [Wie16b] (see Fig. A.2). It focuses on large-scale applications and computes the productivity for a varying number of compute nodes, system lifetime or person-days needed for application tuning. For example, comparisons as presented in Figures 2.2a to 2.2d can be generated. Furthermore, the tool visualizes the results in 2D or 3D figures or exports them in CSV format.

Matlab Model The full complexity of the productivity model introduced in Sect. 4.1 is implemented with Matlab in a modular way. It also separates application-dependent and system-dependent data so that any kind of combination can be applied comfortably. It can further be easily adapted and extended by programmers, e.g., by implementing different performance models for runtime prediction or by analyzing its uncertainty. Finally, Matlab can be used to directly illustrate the productivity results in any conceivable way.

Part B.

Total Cost of Ownership of HPC Centers

TCO is a torch which helps shine
a light into the darker areas of
the cost puzzle.

ANDREW JONES, NAG, AND
OWEN THOMAS, READ OAK
CONSULTING

5. Motivation

Total costs of ownership play a crucial role in assessing the worth of HPC centers, especially given as input for the productivity figure of merit.

Typically, TCO of data centers are distinguished between capital expenditures (Capex) and operational expenses (Opex). Capex describe upfront investments that are amortized over a certain time span and include construction cost of a data center or purchase cost of servers [BCH13, p.91]. Respectively, Opex cover recurring costs caused by running the equipment including electricity, maintenance or salaries of personnel [BCH13, p.91]. TCO investigations are pervasive with respect to data centers since industry businesses strive for minimizing their costs. For instance, Facebook that spent roughly \$4.50 billion for data center equipment and related costs in 2015 [Ovi17], has built a warehouse at the Arctic Circle in Sweden to lower its energy costs [Har17].

Although reduction of costs is naturally also beneficial for HPC university centers, they are mainly driven by their funding agencies who allow a fixed investment usually in the order of tens of million euros. German Tier-1 HPC systems are funded halve by the federal government and halve by the German states with an approximate investment of 400 Mio. € between 2008 and 2017 [Wis15a, p.23]. For German Tier-2 HPC centers, the federal funding instrument of research buildings for institutions of higher education supports an HPC funding program (in German: *programmatisch-strukturelle Linie Hochleistungsrechner*) that promotes the proportional funding by federation and states. From 2007 to 2015, government and states invested roughly 120 Mio. € into HPC (compare Tab. 5.1) [Wis15a, p.24]. For example, the Karlsruhe Institute of Technology were funded with roughly 26 Mio. € for the supercomputer *ForHLR* in 2013 [Wis12], Johannes Gutenberg University Mainz with 8.7 Mio. € for *Mogon II* in 2015 [Wis14], and RWTH Aachen University with 22 Mio. € for *CLAIX* in 2016 [Wis15b]. For the potential follow-up funding concept in the context of the German national

Table 5.1.: Tier-2 HPC funding in Mio. € and according to the funding instrument of research buildings, taken from [Wis15a, p.24].

2007–2010	2011	2012	2013	2014	2015	Sum
13.23	1.40	24.40	28.54	27.26	21.36	116.19

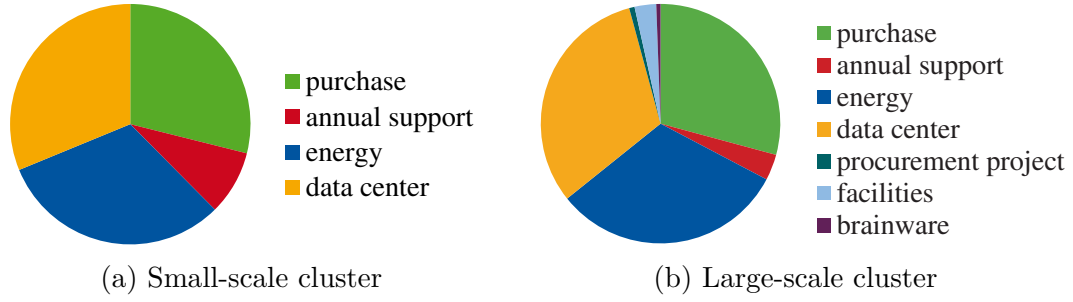


Figure 5.1.: Sample TCO shares for one year according to [JT16] with one-time costs amortized over 6 years.

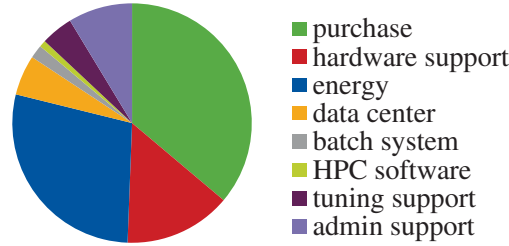


Figure 5.2.: Sample TCO shares for one year according to [BaMI12]

high-performance computing initiative (in German: *Nationales Hoch- und Höchstleistungsrechnen (NHR)*), it was recommended to increase investments by $2.5\times$ compared to current funds [Wis15a].

The German advisory council on scientific matters [Wis15a] recommends these vast investments to provide an incentive to open up HPC systems to user groups outside the respective sponsoring federal state in order to serve the increasing demands for computational power due to more science being based on simulations or big data analysis. Giving further reasons for the rise of investments, they suggest to include factors such as energy costs and personnel costs into TCO and NHR funding. Energy costs have pushed themselves to the fore with HPC exascale objectives and are seized by the ubiquitous power wall. With continuous technological progress, the reduction of investment cycles for next generation hardware makes sense since acquisition costs of novel systems with improved energy efficiency amortize energy costs of previous system generations [Wis15a]. Investments into personnel are meant for educating and financing HPC experts that are able to maintain existing scientific HPC codes and adapt them for new HPC systems [Wis15a].

Depending on the HPC manager’s perspective, political external conditions, the kind of HPC cluster and infrastructure, impact factors on TCO are presented with different amounts of contribution. I pick two samples from Jones’ and Owen’s SC16 tutorial [JT16] where they present their rules of thumb for TCO modeling in

HPC, based on UK data. They illustrate these by spreadsheets containing different costs as percentages of acquisition cost over several years of the cluster’s lifetime. Transferring these cost numbers into percentages of TCO and amortizing all one-time costs over 6 years, I derive TCO shares for a small-scale cluster and large-scale cluster as depicted in Fig. 5.1. Small-scale cluster shares are based on purchase costs set to 3.5 Mio. € and, large-scale cluster shares on 12 Mio. €. By purchase cost, Jones and Owen refer to the capital cost spent for machinery. Annual support covers hardware warranties, and hardware and software maintenance. Energy costs are based on electricity consumed, cooling and the center’s PUE. With data center costs, Jones and Owen mean the actual building and space of a data center. Multi-fold factors contribute to that, e.g., construction cost amortized over 20 years, depreciation, facility maintenance, rack footprints and floor utilization. It is noteworthy that Jones and Owen ignore costs for internal support staff here. For large-scale systems, they reason about a “more involved installation phase with bigger impact on facilities”. In addition, they put into the equation costs for the procurement project itself and assume porting (brainware) efforts for “moderate codes with a reasonably portable infrastructure” to be roughly two person-years. For both of these setups, data center costs, purchase costs and energy are the biggest annual contributors to TCO. Another example is taken from Bischof et al. [BaMI12] and presented in a similar manner. Here, data center costs mean build costs and hold a much smaller TCO share than in the previous examples. Instead, Bischof’s TCO breakdown accounts for administrative staff and also personnel that is responsible for training, documentation performance analysis and code tuning.

As already evident from these three small examples, TCO needs to be modeled in dependence of various impact factors such as the number of compute nodes determining purchase and maintenance costs and the corresponding energy consumption.

6. Related Work

TCO models and TCO quantifications are prominent in (scale-out) data centers whose economic goal of cost reduction is of high priority. The Uptime Institute [KBT⁺07] states as difference between data and HPC center that a “general-purpose commercial data center” has lesser footprint for servers and a greater footprint for storage and networking than a “facility housing HPC analytics applications”. Scale-out data centers such as Google, Amazon, Facebook or Microsoft may also exceed the number of servers of an HPC center. I further distinguish data and HPC centers by their main purpose in terms of applications. While data centers focus on web services or social media with mostly similarly-formed requests, HPC centers are usually exploited by a variety of applications that target on scientific output. Typical metrics for cost estimations in data centers are \$/W or \$/ft² (square feet of floor space). Most TCO studies are published as technical reports or webpages on behalf of, funded or directly conducted by vendors that use TCO comparisons to promote their particular hardware or software solution. Since they apparently want to present their product in the best possible light, caution is advised, especially with an easily-tunable TCO model that may only account for a certain portion of costs. For example, the TCO comparison of Birst Inc. [Bir13] between their cloud-based business intelligence (BI) solution and a traditional and open-source BI solution focuses on presenting costs as manpower costs with huge cost reductions. Similarly, MicroStrategy [Mic08] even tops this by only stating to look at TCO, but actually only emphasizing differences in their BI approach to smaller-scale solutions. Hewlett Packard Enterprise [Hew17] further provides a TCO calculator that directly suggests their right product for the denoted use case. Such calculators are also common as search interfaces for the best cloud provider since the quantification of cloud customers’ costs are straightforward to quantify (see also Sect. 3.5.3). The company UberCloud [Gen16] has even developed a TCO service to help customers judge the benefit from moving from on-premise to cloud hardware. In a sample study, it computes the TCO of a on-premise cluster on cost per hour and compares it to cloud-based solutions.

Contrarily, investigation of TCO models in the context of HPC centers is mostly left out from research. Especially, the quantification and optimization of TCO components is scarcely covered.

6.1. Data Centers

A general introduction to data center facilities is given in Gough et al. [GSS15]. They also provide a high-level view on data center TCO by illustrating some cost factors such as building, infrastructure and IT equipment, and explain the meaning of the metric “performance per watt per dollar”. Another book on data centers [BCH13] covers cost models with more detail. They define Capex and Opex and focus on the physical infrastructure in terms of building costs, server costs, depreciation and servers’ power and cooling costs. They propose to use “cost per critical watt” as TCO metric where “critical watt” accounts for reliability and redundancy factors of the IT equipment. Furthermore, they show TCO breakdowns of three case studies: a classical data center where capital costs account for 78 % of monthly TCO, a data center with commodity-based lower-cost servers and higher power prices where energy costs sum up to 36 % of TCO, and a partially-filled data center with 50 % utilization where data center costs dominate. Finally, they also compare data center costs to public cloud costs. A third high-level introduction is provided by IDC’s white paper [PG07a] including a seven-step process for demonstrating the business value of IT projects to managers. One step suggests to establish business and performance objectives, e.g., by measuring the financial impact using TCO metrics. IDC identifies 40 to 50 impact factors and classifies TCO components by hardware, software, IT staff, services, and user productivity, i.e., server downtimes. Based on 300 interviews, they present a typical three-year server TCO breakdown where staffing contributes to roughly 60 % of TCO.

To promote a certain feature or design of the data center, various works use relative TCO in terms of cost savings presented in percentages, or just quantify (selected) differences in costs. For example, Luo et al. [LGS⁺14] investigate different memory designs and reliability techniques and compare cost savings (in percentages) of memory and server hardware with respect to a “typical” server without specification of absolute costs. Malone et al. [MVB08] from Google and Hewlett Packard specify (only) energy cost savings of a new blade design with an improved system airflow compared to traditional 1U servers. They present energy savings with respect to server fan power and power of computer room air conditioners as percentages, translate these differences into dollars, and denote the price per kWh. Vishwanath et al. [VGR09] by Microsoft look at the performance and reliability of containerized data centers in a service-free design. They specify total costs as component cost plus power cost and provide models for both parts. To derive the component costs, they model MTTF and performance by Markov chains and Markov Reward models, and use these to compute the number of servers needed to provide the corresponding level of reliability and performance. They illustrate their simplified TCO breakdown for three case studies, i.e., servers with HDDs, with RAID and with SSDs. Another comparison is conducted by Intel in their white paper [PCGL07] on high-density vs. low-density scale-out data centers.

They argue against the watts per square foot (W/ft^2) metric due to the ambiguous reference of square foot and rather propose to use cost per server or cost per kW. Finally, they present major cost differences in dollars between these two approaches and, hence, argue for high-density spaces. An IDC study [QFSH06] takes data from Hewlett Packard and compares blade server designs to rack-optimized servers setting up numerous cost factors. They include their assumptions on hardware, management software, initial deployment and data center setup in detail, but only present costs as savings. As a quick tool in the planning phase, the Uptime Institute [TS06] introduces a TCO model that is based on the power density of the data center and its electrically active floor area: $\text{ft}^2 \cdot \$/\text{ft}^2 + \text{kW} \cdot \$/\text{kW}$.

A more comprehensive way of modeling TCO has been approached by the University of Cyprus. Their works [HSSS11][HKS⁺13] introduce EETCO a “tool to Estimate and Explore the implications of data center design choices on the TCO”. They start with defining TCO as sum of costs for infrastructure, server, network, power and maintenance, and then refining the corresponding TCO components, e.g., by including MTTF and an estimation of cold spares. In their case study, they compare the normalized TCO breakdowns of high-performance servers and low-power high-density servers. An extension of EETCO is undertaken by Nikolaou et al. [NSNK15] for modeling DRAM failures and DRAM error protection techniques. They comprise models for performance degradation due to the chosen error-correcting code (ECC) technique or DIMM costs. This EETCO tool was further applied to argue for a new scale-out processor design [GHLK⁺12] where the authors plot normalized TCO across different chip-level designs and approach cost effectiveness by computing performance per TCO with performance depending on user instruction and processor frequency. Another cost model that focuses on the physical cost is introduced by Karidis et al. [KMM09]. They define cost as sum of infrastructure, floor space, server and electricity costs, express all components in $\$/\text{kWh}$ and normalize them with the server’s utilization factor. Here, infrastructure costs relates to the Tier-level of the data center and its depreciation, server costs to their hardware costs and depreciation, and electricity costs to the server utilization and the price per kWh. They extend their cost model with a labor cost component and compare one m -socket server to m one-socket servers using queuing theory. In an HP Labs technical report [PS05], the authors briefly explain a typical data center design, suggest a cost model and use it for arguing on the “right provisioning”. The focus of their cost model is similarly to Karidis et al. on the physical space, power and cooling. Space costs include real estate and the floor’s occupancy, power costs are expressed as “burdened costs of power delivery” including direct electricity costs, but also penalties for non-optimal operational designs on the data center level. Similarly, cooling costs represent burdened costs of inefficient cooling systems. However, the authors realize that a real TCO computation also needs to account for operational costs, i.e., labor costs, software licenses and depreciation, which is simply expressed as cost per rack.

6.2. HPC Centers

Contrary to the elaborative work on data centers, research on TCO models and TCO quantification in the HPC domain is less prominent. Ludwig’s SC12 invited talk on the costs of HPC-based science [Lud12] and the follow-up Birds-of-a-Feather session at SC13 [LRA13], where Ludwig, Wienke, Reuther, Apon and Joseph presented on HPC cost-benefit quantifications, showed that the HPC community expresses interests in that topic, though.

While for data centers, \$/kW or \$/ft² are still common, Hsu et al. [HFA05] reason that the HPC-popular performance-power metric is insufficient to express reliability, availability, productivity, and TCO. They show that ratio formulas where performance is raised to the power ≥ 2 put a (too) strong focus on performance. Unfortunately, they focus on the Linpack Flop/s performance metric and do not propose any other metrics, but resign and still use Flop/s per watt for their evaluations. Other works that cover TCO in HPC rely on real cost computations and results in dollars or euros.

In the past, most TCO models have been introduced in the context of productivity metrics as discussed in Sect. 3. However, the majority of these works lack a real-world quantification of TCO components. The most comprehensive approach is taken by Murphy et al. [MNV06] who provide a table of initial guesses for their TCO formulas as introduced in Sect. 3.3.3. The Uptime Institute focuses on HPC facilities in their work [KBT⁺07] and presents a great variety of factors in a spreadsheet together with their assumptions on how to quantify these. However, since their origin comes from data centers instead of HPC centers, the granularity of possible optimizations on the application level or job scheduler level is completely missing. With special focus on storage systems, Kunkel et al. [KKL14] model the costs per job incorporating compute costs, storage costs and archiving costs, excluding manpower costs. Compute costs are determined by the number of jobs, the number of systems and the job’s compute time. Storage costs incorporate the job’s (accessed) data volume over time with respect to the file system’s capacity, throughput and metadata rate. Third, the archive costs depend on the tape media, and the checkpoint costs are driven by writing to the storage system while the rest of the system is idling. In addition, each cost component contains certain running costs that account for amortized purchase and power costs. Kunkel et al. apply this cost model to two real-world case studies from weather and climate simulation at the German Climate Computing Center (DKRZ) and further evaluate different concepts to handle storage costs, i.e., recomputation of results, deduplication and compression of data.

7. Modeling Total Cost of Ownership in HPC

The millions of funding invested in HPC centers demand informed decisions for procurements that are accompanied by an appropriate TCO model. This TCO model needs to cover all HPC-relevant components and, especially, needs to provide baselines for their quantification and estimation in the tendering process.

In Sect. 7.1, I recap the foundations of my TCO model introduced as part of the productivity metric in Sect. 4.3 and elaborate on its composition for single-application and multi-application setups. The discussion on the model's parameters in terms of quantification and predictability is presented in Sect. 7.2.

The single-application TCO approach presented here is mostly based on work in Wienke et al. [WaMM13] that covers the TCO of real-world setups including CPU and accelerator systems. The same TCO model was applied to a large-scale case study in Wienke et al. [WIaMM15]. A slight revision of the energy cost component of the TCO model was implemented in our work in Schneider et al. [SWM17].

7.1. Definition of TCO

Recapitulating the previously-introduced model, total costs are divided into one-time and annual costs. In addition, the costs are denoted per node and per node type, and are either dependent on the system or the applications. The basic formula is given in Equ. (4.13) and HPC-relevant TCO components in Tab. 4.1.

7.1.1. Single-Application Perspective

Combination of the TCO formula and all components into one equation yields:

$$\begin{aligned}
 \text{TCO}(n, \tau) &= [C^{ot,n} \cdot n + C^{ot,nt}] + [C^{pa,n} \cdot n + C^{pa,nt}] \cdot \tau \\
 &= [(C_{HW}^{ot,n} + C_{IF}^{ot,n} + C_{VE}^{ot,n}) \cdot n + C_{EV}^{ot,nt} + C_{DE}^{ot,nt}] + \\
 &\quad [(C_{HM}^{pa,n} + C_{IF}^{pa,n} + C_{VM}^{pa,n} + C_{EG}^{pa,n}) \cdot n + C_{EM}^{pa,nt} + C_{SW}^{pa,nt} + C_{DM}^{pa,nt}] \cdot \tau
 \end{aligned} \tag{7.1}$$

Here, the components' dependency on node level or node-type level is denoted with superscript n or nt , respectively, and all costs include taxes. The system hardware costs $C_{HW}^{ot,n}$ and the corresponding maintenance $C_{HM}^{pa,n}$ are usually direct costs laid down in the purchase agreement where $C_{HM}^{pa,n}$ is normally given by a percentage p_{HM} from hardware purchase costs. One-time infrastructure costs $C_{IF}^{ot,n}$ can be either direct costs or depreciated over several years and then captured by $C_{IF}^{pa,n}$. The software and compiler costs $C_{SW}^{pa,nt}$ are also generally direct costs.

Costs for setting up the OS and software environment $C_{VE}^{ot,n}$ and $C_{EV}^{ot,nt}$ and their maintenance costs $C_{VM}^{pa,n}$ and $C_{EM}^{pa,nt}$ are mainly determined by the administrators' required total effort AE in person-days and their salaries sl in € per person-day, so that, e.g., $C_{VE}^{ot,n} = AE \cdot sl$. The same holds for the development and tuning effort DE spent for parallelizing, optimizing or porting applications and the corresponding application maintenance effort DM and their corresponding costs $C_{DE}^{ot,nt}$ and $C_{DM}^{pa,nt}$. In addition, development effort DE depends on numerous impact factors as I discuss in Part C. Especially, the (target) application runtime t_{app} that might be reached by additional tuning affects the amount of needed effort:

$$C_{DE}^{ot,nt} = sl \cdot DE(t_{app}, \text{factor}_2, \dots, \text{factor}_k) \quad (7.2)$$

The energy costs $C_{EG}^{pa,n}$ depend on the energy consumption of the workload, the provided electricity costs ec in €/kWh and the center's PUE pue . While the application's energy consumption can be modeled comprehensively, I follow here a cognizable approach for illustration. With my basic assumption that the application is executed during the whole system lifetime, I replace the application's energy by its power consumption co . The power consumption co could be an averaged value over the application's runtime, or modeled with respect to the application's characteristics: Naively, the application can be split into a parallel portion p_{par} that runs with high power consumption leveraging numerous hardware units, and a low-power serial portion $(1 - p_{par})$. This case yields

$$C_{EG}^{pa,n} \sim ((1 - p_{par}) \cdot co_{ser} + p_{par} \cdot co_{par}) \cdot ec \cdot pue. \quad (7.3)$$

While all one-time costs are obviously independent of time, the annual costs may be affected by variations over system lifetime. One impact factor is system availability α since it reduces the (up)time that systems are executing application runs. Therefore, the application-dependent energy costs must account for that by splitting $C_{EG}^{pa,n}$ into an application-dependent component $C_{EG,app}^{pa,n}$ during system availability and a system-dependent component $C_{EG,sys}^{pa,n}$ for system unavailability:

$$C_{EG}^{pa,n} = \alpha \cdot C_{EG,app}^{pa,n} + (1 - \alpha) \cdot C_{EG,sys}^{pa,n} \quad (7.4)$$

where $C_{EG,sys}^{pa,n}$ relates to some power consumption cu in watts during system unavailability. Several approaches can be undertaken to determine cu depending on

which kind of system unavailability is most probable. If the system is unavailable because of downtime, e.g., due to failures, cu amounts to 0. If the system is unavailable due to maintenance, it can be assumed that it is running on low power, e.g., idle power ci so that $cu = ci$. If the system is only unavailable for production runs because of pre-production or full test runs, $cu = co$ could be assumed.

Summarizing, since my TCO model definition targets at procurement processes, I propose a high-level approach with straightforward easily-applicable terms, instead of setting on advanced detailed formulas. If procurement reasoning necessitates further breakdowns, the components of my TCO model can be refined using more specific formulas. For example, the storage cost model by Kunkel et al. [KKL14] could be employed into the infrastructure cost component.

7.1.2. Composition to Job Mix

The enhancement of the single-application TCO perspective to job mixes as stated in Equ. (4.15) to (4.17) follows the HPC center view of procuring, installing and maintaining hardware that suits numerous applications. Because of this, I have chosen to keep the costs of all system-dependent components as they are and only adapt the application-dependent costs, instead of splitting up all TCO components. Hence, for m applications, I assume

$$\begin{aligned} \text{TCO}(n, \tau) = & \left[(C_{HW}^{ot} + C_{IF}^{ot} + C_{VE}^{ot}) \cdot n + C_{EV}^{ot} + \sum_{i=1}^m C_{DE,app,i}^{ot} \right] + \\ & \left[(C_{HM}^{pa} + C_{IF}^{pa} + C_{VM}^{pa} + \frac{1}{n} \sum_{i=1}^m (p_i \cdot n_i \cdot C_{EG,app,i}^{pa})) \cdot n + \right. \\ & \left. C_{EM}^{pa} + C_{SW}^{pa} + \sum_{i=1}^m C_{DM,app,i}^{pa} \right] \cdot \tau \end{aligned} \quad (7.5)$$

instead of

$$\begin{aligned} \text{TCO}(n, \tau) = & \sum_{i=1}^m \left(\left[p_i \cdot (C_{HW}^{ot} + C_{IF}^{ot} + C_{VE}^{ot}) \cdot n + p_i \cdot C_{EV}^{ot} + C_{DE,app,i}^{ot} \right] + \right. \\ & \left[p_i \cdot (C_{HM}^{pa} + C_{IF}^{pa} + C_{VM}^{pa} + C_{EG,app,i}^{pa}) \cdot n + \right. \\ & \left. p_i \cdot (C_{EM}^{pa} + C_{SW}^{pa}) + C_{DM,app,i}^{pa} \right] \cdot \tau \Big). \end{aligned}$$

Thus, the development and maintenance costs for multiple applications are simply aggregated (see Equ. (4.15)). While development costs for single applications could have also been interpreted as time schedule, i.e., elapsed time span needed afore system production, this does not hold for the multi-application setup anymore. One reason is that multiple developers are potentially simultaneously

working on the applications. Energy costs for the whole job mix should also account for system unavailability as illustrated in Equ. (7.4). Here, the application-dependent components aggregate according to the capacity-based percentages (following Equ. (4.17)) and the system-dependent components remain as they are.

7.2. Discussion & Quantification of Components

The given definition of my TCO model suggests various components. They represent in number a tradeoff between model accuracy and over-parametrization. The latter may evoke a huge amount of error sources and extra effort to specify required values. Nevertheless, these TCO components can be further refined or reinterpreted if necessary. Some use cases are discussed below.

From the experiences gathered at RWTH Aachen University, I illustrate how TCO components can be quantified and predicted in the context of a procurement process. An overview is shown in Tab. A.5.

7.2.1. Hardware Purchase Costs $C_{HW}^{ot,n}$

Hardware purchase costs are one of the major contributors to total costs (compare TCO breakdowns in Fig. 5.1) and also play an important role in productivity results as illustrated by Schneider and me [SWM17] in case studies on different operational node concepts. Although purchase costs seem to be fixed, special deals might be arranged in big procurement processes that are constrained in investment.

Quantification & Predictability Since purchase costs are direct costs, a reliable source of information is needed for reasonable quantification, usually the vendor or reseller. For small procurements, price lists from online shops may give a first impression. For big procurements, personal interviews with vendors are inevitable. Since these kinds of interviews and meetings are common before the tendering process, purchase costs can be well estimated. To give a rough indication, a standard CPU-based server for HPC purposes is in the order of 5,000 €.

7.2.2. Hardware Maintenance Costs $C_{HM}^{pa,n}$

During the hardware maintenance time period that is contractually laid down between HPC center and vendor, broken hardware components get replaced with

no additional costs. The corresponding hardware maintenance costs are usually denoted as percentage p_{HW} of the net hardware purchase costs.

My TCO and productivity model currently sets the contractual maintenance period according to the system lifetime. Thus, modeling the system operation for 20 years assumes maintenance support for 20 years with steady percentage p_{HM} . However, as indicated in the discussion on the system lifetime parameter (see Sect. 4.4.2), longer system operation increases the chance of broken hardware parts. Therefore, vendors will actually not sign contracts with such a huge time span for economic reasons. I will include a machinery breakdown rate, and maintenance periods independent from system lifetime in future research.

Hardware Maintenance Percentage p_{HM}

The value of percentage depends on the required service conditions. For example, broken components might be replaced within 24 hours or only within a greater time span; and their replacement might be carried out directly by external staff or the HPC center's administrators.

Quantification & Predictability Similar to hardware purchase costs, maintenance costs or percentages can be quantified through interviews with vendors and, hence, well predicted. Typical values range from 5 % to 15 % of hardware acquisition costs.

7.2.3. Software and Compiler Costs $C_{SW}^{pa,nt}$

Commercial software provided on HPC systems for all users comprises compiler such as Intel or PGI, development environments such as parallel debuggers, performance analysis tools, e.g., Intel Amplifier VTune, or data analysis tools like Matlab. Additional software might be open-source or acquired for special purpose by university institutes and integrated into the cluster environment with restricted access. Fortunately, commercial software licenses do not scale with the number of compute nodes or users, but usually work with floating licenses. Nevertheless, parallel software usually asks for licenses per process. Thus, some software like debuggers might need a higher investment. Although software costs do not relate to the number of nodes, they are still dependent on the compute node type. For instance, GPU clusters may require additional software for accelerator programming, e.g., particular compilers or and analysis tools.

Quantification & Predictability To quantify and estimate the required software costs, HPC managers need to determine the users' requirements on software on the HPC cluster. For that, HPC managers can either interview their users, or rely on experiences from previous cluster installations. From previous years, the number of requests for certain software modules can usually be revealed by cluster statistics and give an indication on useful software packages. Based on this information, corresponding costs can be directly obtained from resellers and vendors. A typical annual amount taken from RWTH Aachen University is 50,000 €.

7.2.4. Infrastructure Costs $C_{IF}^{ot,n}$ and $C_{IF}^{pa,n}$

Infrastructure comprises network switches, cables, uninterruptible power supplies (UPS), file systems, fire suppression installations, building construction and many more components. Most of these parts scale roughly with the number of nodes or can be expressed in a linear relationship. Despite the variety of infrastructure components, the TCO computation should only contain the relevant ones. Here, the existing infrastructure and its reusability mainly determine the accounted components. Occasionally, infrastructure costs may also be covered in a separate TCO calculation when funding is planned for different procurement and tendering processes. Nevertheless, I usually express all costs in absolute values and forgo popular metrics used in traditional data centers, such as \$/ft² or W/ft², since some related works [TS06][PCGL07] show shortcomings of these approaches.

High capital infrastructure costs are often amortized over several years. For RWTH Aachen University, the construction of a building that can accommodate a particular HPC cluster follows this kind of depreciation.

Quantification & Predictability The expenses for most infrastructure components can be quantified and estimated by interviews with vendors and other suppliers. However, if expressed on a per node basis, further assumptions might be needed. I take exemplary the RWTH's building expenses as decomposed in [BaMI12][WaMM13] that account for 7.5 Mio. €. Since this building also hosts offices and visualization facilities, initial costs are assumed to be 5 Mio. €. The building's depreciation over 25 years yields 200,000 € per anno. Since the limiting factor for housing compute nodes and infrastructure is its electrical supply of 1.6 MW, the maximum power consumption per node co_{max} is decisive. It may be evoked and measured by a Linpack run on that hardware. Thus, one possible breakdown of the building costs on a per node basis is given by $0.2 \text{ [M€]} / 1.6 \text{ [MW]} \cdot co_{max} \text{ [W]}$. Again, these parameters should become available on request from the vendors and suppliers. If needed, the maximum power consumption can be estimated by the server's thermal design power (TDP) as the example in Sect. 15.2 illustrates.

7.2.5. Environment Installation Costs $C_{VE}^{ot,n}$ and $C_{EV}^{ot,nt}$

Similar to the infrastructure expenses, the amount of environment setup costs depends on existing solutions for cluster management that can be taken over to new cluster installations. This may include technology enhancements to roll out software to all compute nodes or the center-specific configuration of a batch scheduler, e.g., with respect to accounting rules. One-time environment configurations also incorporate the initial installation of the OS and the creation of a security concept. Fortunately, most of these setup steps rather belong to the whole cluster (type) instead of scaling with the number of nodes. However, if the HPC center's administrators have not worked with the given hardware type before, additional costs may arise. For example, the first commission of a GPU cluster means additional GPU driver installations, testing, and special batch system integration.

The two main components of the environment installation costs, i.e., administration effort and corresponding salaries, are discussed separately in the following.

Administration Effort AE

Effort (in person-days) needed by administrators to setup the cluster environment depends on existing solutions aforementioned, but also on their experience and the purchased hardware and software configuration.

Quantification & Predictability Since administration effort depends on numerous impact factors, its quantification and estimation is challenging. Quantifications mainly base on experiences with previous installations. Estimations could be derived by a simplified Delphi Process known from SE where administrators estimate their needed effort independently on a work breakdown structure and then discuss their assumptions in a plenum.

Salary sl

Most personnel at German university HPC centers are employed in civil service.

Quantification & Predictability Quantifications can be founded on recommendations on annual personnel rates from the German Science Foundation [Ger17] that yearly amount to, e.g., 47,100 € for non-academic staff and 63,300 € for doctoral researchers (or comparable) in 2017. Furthermore, the European Commission's CORDIS [Eur14] suggests having 210 working days when excluding vacations and sick days. Overall, this results to 224.29 € per person-day for non-academic staff and 301.43 € per person-day for doctoral researchers. Alternatively, if fixed staff is attributed to the administration services, their direct salaries can be taken.

7.2.6. Environment Maintenance Costs $C_{VM}^{pa,n}$ and $C_{EM}^{pa,nt}$

As annual component of the environment upkeep, maintenance tasks include testing, monitoring and reporting, as well as security maintenance with regular updates, software support, or migration activities. In addition, administrative maintenance efforts also depend on the targeted system availability α , especially if availability is not allowed to drop below a threshold.

Quantification & Predictability For a case study at RWTH Aachen University, a quantification of annual administration effort was extracted from interviews and is given by four administrators (FTEs) managing the cluster in 75 % of their time. If administration times must be expressed on a node level, the overall effort can be divided by the maximum number of compute nodes in the cluster. Nevertheless, effort estimations are still challenging, especially if no prior cluster installation providing empirical values exists.

7.2.7. Development Costs $C_{DE}^{ot,nt}$

Development costs arise for every application that needs to be parallelized or adapted for the hardware architecture selected in the procurement process. The corresponding needed amount of development costs and effort determines the usefulness of the cluster for the university's researchers.

Development Effort DE

Development costs are defined by software development efforts and corresponding salaries analogously to administration costs. My definition of development effort introduced in Sect. 8.1 embraces major HPC-related activities. As I also discuss in Part C, development effort is affected by numerous factors that are not easily tangible and quantifiable. A simplified model such as the Pareto Principle representing the 80-20 rule (Sect. 8.2.2) might express the relationship between application performance and effort needed to achieve this performance (indicated in Fig. 10.1). However, comprehensive relations should also account for the developer's skills, kind of application, targeted hardware or available tools landscape.

Quantification & Predictability Thus, to tackle these challenges, I propose a methodology for HPC effort estimation in Part C. As little data is publicly available on effort spent for completion of real-world HPC projects, I take an example from RWTH Aachen University. This case study covers porting and tuning of the real-world simulation package *ZFS* for GPUs and yielded 75 person-days of reported effort (see Part D).

7.2.8. Application Maintenance Costs $C_{DM}^{pa,nt}$

Comparable to development costs, application maintenance costs are aggregated over all applications running on the cluster. Here, maintenance efforts relate to HPC activities only, e.g., testing and verifying new compilers or adding performance optimizations to the code. Maintenance and modification of the domain-specific base codes are not part of the TCO equation of HPC centers.

Maintenance Effort DM

The investigation of maintenance effort in mainstream software projects is covered in cost models in SE. However, these approaches rather focus on repairing defects or adding customer support, and, thus, are not assumed to suit HPC-related application maintenance very well. An analogous discussion on the applicability of SE software cost estimation to HPC is presented in Sect. 9.2.1.

Quantification & Predictability Since little research is conducted on application maintenance in HPC projects, I cannot present any sample quantifications. The creation and validation of a corresponding model is out of scope of this work and therefore, unfortunately, developers are left with their instinct for estimation of this component. However, if maintenance means further performance tuning, my methodology for effort estimation can be applied by taking the parallel code version as starting point.

7.2.9. Energy Costs $C_{EG}^{pa,n}$

Costs for energy massively contribute to total costs as depicted in Fig. 5.1.

While I have introduced a simple power model in Equ. (7.3), the TCO formulation can also incorporate more complex power and energy models. With the increasing importance of power optimizations, energy efficiency of supercomputers has been emerged to its own wide research field. With that noted, I refer for more comprehensive models to recent research activities.

Though, I still like to emphasize that power models may differ widely with respect to the level and technique of the investigated or applied power optimization. For example, power consumptions of applications may be influenced by the HPC center's power shedding or shifting philosophy or by service contracts with electricity providers that allow dynamic pricing. The extent of these techniques is surveyed by Patki et al. [PBG⁺16] in German and U.S. supercomputing centers.

Furthermore, my current approach of power evaluation is based on consumption per application and compute node with the intention of being able to capture particular energy-efficient hardware architectures and power optimizations in programs. Consequently, the power consumption of each application is considered proportionally to their capacity shares. In other setups, planning energy budgets on granularity of the complete HPC center might be more appropriate. Potential reuse of the cluster's waste heat for heating offices adds further complexity to energy cost calculations (also compare cash flows in Sect. 4.5).

Power Consumption *co*

Node-level power consumption is the cost factor that is most easy to influence by HPC managers and developers, e.g., with the choice of an energy-efficient hardware architecture or application-based energy optimizations.

Quantification & Predictability The quantification of an application's power consumption can be undertaken by measurement using power meters or, if not available, approximated by chip-internal counters, e.g., RAPL on Intel servers. Typical real-world values for Intel Broadwell two-socket servers are 220 W for serial code portions and 350 W for parallel code portions. While aforementioned power models may be used for comprehensive estimation of power consumption, my simplistic approach only requires the prediction of the application's runtime characteristic, i.e., serially and in-parallel executed parts, and the corresponding (averaged) power consumptions. The former is solved with performance models in the context of runtime estimation (see Sect. 4.4.3). The latter could be tackled by, e.g., reference measurements from vendors that trace power consumption for 'arbitrary' codes leveraging single-core and multi-core performance. At best, vendor benchmarks are not completely 'arbitrary' but rather approximate some of the application's general characteristics such as vectorization or data access patterns. An example on the prediction of this value can be found in Part D.

Electricity Costs *ec*

Electricity costs are laid down in contracts with electricity service providers and depend on the agreed pricing structure.

Quantification & Predictability Given available options from the electricity service providers, electricity costs are directly quantifiable and predictable. Today, German HPC centers may pay approximately 0.15 €/kWh to 0.18 €/kWh on average in a fixed pricing model.

Power Usage Effectiveness *pue*

The PUE value is set for the whole HPC center and strongly relies on infrastructure technologies and setups, including cooling techniques.

Quantification & Predictability As modern HPC centers are usually equipped with power distribution units at numerous infrastructure components, the input and used power can be measured and quantified. HPC centers typically operate at a PUE in the range from 1.1 to 1.8. With a given specification of the infrastructure, the PUE can be estimated from experiences by the HPC managers or vendors.

7.2.10. Other Parameters

To find the line between few and many TCO parameters and between an easily-understandable and complex TCO model, I left out some components that other HPC managers might wish to include. Here, I exemplary introduce some of these for future discussion.

Reuther and Tichenor [RT06] suggest to include time for training users with respect to (efficient) use of HPC resources. Similarly, daily user support answering tickets in terms of parallel software usage can be incorporated.

Jones and Owen [JT16] estimate 2 % to 10 % of purchase costs for the procurement project itself depending on the size of the cluster (compare Fig. 5.1b) that might be included for TCO calculations. With the operation of an HPC cluster, funding sources ask for scientifically-based review processes of compute-time applications. Thus, corresponding project proposals are reviewed, and effort spent for that adds costs to TCO.

Finally, the whole commissioning process has not been considered so far. Acceptance tests and Linpack runs increase total costs without producing any scientific result. Furthermore, an HPC center might run pre-production phases that restrict usage to very few HPC projects with the purpose to extensively test the environment. After the system's lifetime, it also needs to be decommissioned potentially including data migration and removal.

Future studies and sensitivity analyses will reveal the need for inclusion or adaptation of TCO components.

Part C.

Development Effort Estimation in HPC

Prediction is very difficult, especially about the future.

NIELS BOHR, DANISH PHYSICIST, 1885-1962

8. Motivation

Effort spent on developing parallel, high-performance applications and corresponding manpower costs are an important part of total ownership costs and thus HPC productivity. Exemplary, for general IT investments, IDC identifies IT staff as one major TCO category and quantifies staffing costs with 60 % of the three-year TCO for a typical server based on over 300 interviews [PG07b]. With respect to HPC programming time, Bailey and Snavely [BS03] assume that software costs make up 80 % of total system cost (here software, hardware and maintenance) for some HPC applications. One step further, Bischof et al. [BaMI12] show that investing into HPC experts who tune cluster applications pays off in terms of HPC ROI.

The importance of software development effort will even rise in future: Large-scale HPC systems with their ever increasing demands for computational power face the challenge of meeting electrical power and budget constraints as discussed in Sect. 3. Given these circumstances, technological advances yield increased hardware and software complexity that applications have to deal with. As a direct consequence, development effort increases for parallelizing, tuning and porting. The rise in importance of development effort has also been recognized by the ZKI and the German advisory council on scientific matters: In their recommendation reports for future developments in the field of HPC, they propose to include brainware and methodological expertise into the governmental funding structure.

Hence, it is essential to include development efforts into software project management and especially into TCO and productivity models where it must be estimated in the best possible way.

8.1. Definition of Development Effort in HPC

Since the term *effort* is ambiguous in usage and literature, I clarify and present my interpretation of effort including reasons for choosing this approach.

In SE, software metrics are distinguished between product and process metrics. “Product metrics relate directly to the result of a software development process” [BD08, p. 208] and measures features of the product such as size, functionality, complexity or reliability [FB14, p. 90]. Due to their character, various product metrics can be easily quantified, often even in an automated fashion [FB14, p. 94].

Table 8.1.: Definition of development time: includes and excludes.

Includes	familiarization with the (serial) domain code
	familiarization with the parallel computer architecture
	familiarization with the parallel programming model
	familiarization with needed tools
	adding parallelism to code
	porting codes to other architectures or parallel programming models
	performance optimization & tuning
	debugging
	testing correctness
Excludes	batch system preparation
	sheer waiting time on scheduling/ results from a batch system
	writing of project proposal for compute-time application
	writing of base code
	application maintenance

Because of this, much focus has been put on this kind of metrics so far and I further discuss them in Sect. 9.1. Instead, “process metrics relate to the software development process, comprising the activities, methods, and standards used” [BD08, p. 225] and cover time, effort, quality or cost and, therefore, are of strong interest for HPC decision makers. Thus, I refer to development effort as process metric. In more detail, I equate development effort with software development time since development time (in person-hours or person-days) is the measure that is valued with a salary. The latter is relevant for incorporation into my TCO formula.

8.1.1. Development Time

The metric *development time* has been used in several HPC-related effort evaluations [HBVG08][ESEG⁺06][HCS⁺05][CES12] including some of my previous publications [WaMM13][WlaMM15]. It has been applied to different HPC use cases such as adding parallelism to a given code, tuning it, porting it, as well as comparing the effort using various parallel programming models, algorithms or tools. Sadowski and Shewmaker [SS10] summarize briefly the coverage of time-based metrics from their literature review on usability issues in HPC. However, no clear comprehensive definition exists that specifies activities covered by the development time metric.

For this work, my definition of development time includes all efforts related to parallel development and performance optimization of scientific applications as listed in Tab. 8.1. As one typical use case, I assume that a base code version

of the application under investigation already exists which is usually developed and maintained by domain scientists like biologists, physicists or chemists. For instance, this base version can be serial code that gets parallelized and tuned for performance, or it can be a previously-parallelized code that gets further optimized or ported to a different parallel hardware architecture. I call the union of all these efforts and tasks *HPC activities*. I further focus on use cases where HPC activities are exploited by HPC experts or HPC-experienced staff in collaboration with the domain scientist, e.g., employees at HPC centers or corresponding cross-sectional groups [JAR17b]. Thus, I include time spent on familiarizing with the domain code into my definition of development time. Moreover, especially with the emergence of novel parallel computer architectures and parallel programming models, the time needed to get to know these concepts must be considered since it contributes to the setup costs for an HPC center. This time may also incorporate the familiarization with tools for debugging or performance analysis that the HPC-experienced developers have not used so far or that are newly released. The obvious HPC-related activities such as parallelization, porting and tuning are accompanied by a testing phase that includes debugging and correctness checks. Both can take much more time than in general software engineering. For instance, Kepner [Kep04a] lists as testing time 34 % to 50 % of the total time spent for six HPC National Aeronautics and Space Administration (NASA) projects. This is not surprising given the parallel nature of HPC programs and, hence, the complexity of handling up to thousands of processes in a single debug session. The difficulty is further increased by the constraint of having often only limited software licenses for process debugging or relying on the batch system for interactive sessions. Moreover, correctness of a code can be defined on different levels, so that the domain scientist has to specify the acceptable accuracy of the results generated by the (parallel) code. Again, achieving the required accuracy with parallel code may be tricky due to different instruction set architectures (ISAs), order of numerical operations or random number generators. Finally, I also include time needed to prepare jobs for the batch system.

While batch system preparation is included in my definition of development, I do not separately account for the waiting time on batch results (compare Tab. 8.1) since I assume that this time can be used efficiently for other tasks of the HPC expert. Especially, batch waiting times in German HPC centers that implement a scientific review process with predetermined compute-time application periods and dedicated project compute time should be moderate. For example, the average waiting time for JARA projects at RWTH Aachen University in 2015 amounted to approximately 15 hours per job. Although this scientific review process yields additional effort to write project proposals for the application of compute time, I exclude this effort from my development time definition for now. However, in future, this could be added for a more comprehensive productivity investigation of HPC centers. Third, as explained above, I assume that HPC developers can

built upon an existing code base. Therefore, I also exclude the effort spent writing this code base, especially, since this effort is often unknown or contrarily might be covered with general SE techniques. While effort for application maintenance is also important to consider, it should be investigated separately and is, thus, outside the scope of this work.

Completing my definition of development effort, I define the following manpower units for quantifications in effort studies: One person-day consists of 8 hours, and one person-week of 5 person-days. According to the European Commission’s CORDIS [Eur14], I assume a total of 210 person-days per year accounting for weekends, vacation and sick days. This gives an average of 17.5 person-days per month.

Summarizing, the definition of development time introduced above fully accounts for the complexity and difference of developing parallel software compared to traditional software as investigated in Sect. 9.2. Nevertheless, the development time metric entails three main risks: First, it is difficult to gain measures for development time since developers usually do not track their effort, especially not on a level of detail that gives insight into the parallel-only part. Second, if developers track their effort, this is often done on a base of manual reports. Studies that compare reported efforts to observed efforts, i.e., values obtained by an external observer accompanying the developer, show that software developers over or under report their effort. For example, Perry et al. [PSV95] find an average over reporting of 2.8 % across 20 subjects for general software development, where the error of reported efforts varied between roughly -13% (under reporting) and 28% (over reporting). In HPC development, the two-subject study of Hochstein et al. [HBZ⁺05] shows an over reporting of 13% for a graduate student, and a small under reporting of 1% for a professional programmer. Thus, even if efforts are tracked by developers, uncertainty in the collected data must be considered. Third, since development time can be used for a variety of HPC use cases, it is also affected by numerous factors such as the developer’s capability or the kind of parallel programming model. To tackle all these challenges, I setup a methodology that tracks the impact factors on development effort, incorporates them into an effort estimation model and improves on the accuracy of collected data.

8.1.2. Time to Solution

In connection with effort investigations in HPC, the metric *time to solution* is often in discussion. However, time to solution is not clearly defined either. Usually, it comprises some development time and execution time component. Therefore, it has also been announced as a productivity metric from the user’s perspective (see Sect. 3). In Bailey and Snaveley [BS03], time to solution is defined as combination of “1) time devoted to programming and tuning; 2) problem set-up time and

grid generation; 3) time spent in batch queues; 4) execution time; 5) I/O time; 6) time lost due to job scheduling inefficiencies, downtime, and handling system background interrupts; and 7) job post-processing, including visualization and data analysis”. Consequently, my approach on development time is orthogonal to the time to solution metric and can be fed back into that metric.

8.2. Estimation of Development Time

Research on development effort estimation is a broad field and well established in software engineering where it is referred to as *software cost estimation*. I introduce corresponding techniques used in traditional SE in Sect. 9.2. In HPC, little research has been conducted in this area though.

8.2.1. Current Techniques in HPC

A substitute for the investigation of software development effort in HPC has been taken up using source lines of code as numerous HPC publications show (see Sect. 9.1.1). There, SLOC are mostly used to compare the ease-of-use of various parallel programming models, e.g., OpenMP and MPI. The main reason for the prevalence of the SLOC metric is that it is easy to quantify. However, the application of SLOC is also controversially discussed as I show in the following sections. Another downside using SLOC as replacement for effort is that it cannot directly be applied in any TCO and productivity equations, but needs an additional conversion factor in order to compute software costs. Furthermore, no research has been conducted yet on how to predict SLOC in the area of HPC. Thus, its usage in a productivity model with predictive power is very limited.

Through discussions with different HPC managers and HPC software developers, I find that some of them use *expert judgment* as estimation technique in software project management. Expert judgment means that an individual or group of experts (at best) with different backgrounds estimate effort of individual tasks that are then combined in a bottom-up aggregation [McC06]. However, expert judgment also faces the challenge that software and hardware complexity is increasing. Novel parallel computer architectures and programming models get released frequently, so that obtaining expert knowledge is an contemporary issue and prediction for even non-released HPC setups seems difficult.

8.2.2. Performance Life-Cycle

An additional estimation technique that can help supporting expert judgment is based on parametric models (for example COCOMO II described in Sect. 9.2.1).

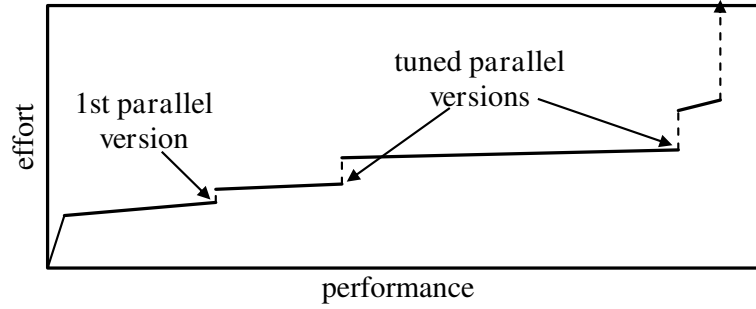


Figure 8.1.: Exemplary performance life-cycle with different milestones.

Since HPC activities target at performance-critical environments, my estimation approach works with the relationship of performance to effort and is referred as *performance life-cycle*.

One popular effort-performance relationship is given by the 80-20 rule. It suggests that 20 % of effort will result in 80 % of performance and that 80 % of effort are needed to achieve the remaining 20 % of performance. I show the applicability of this (simplified) rule for a real-world setup by modeling it by a Pareto distribution [New05] in Wienke et al. [WlaMM15]. However, experiences also show that this simplified rule does not always suffice since an effort function depending on performance (and other impact factors) commonly follows a more complex relationship. An exemplary performance life-cycle is illustrated in Fig. 8.1 and shows dependency on different milestones represented by ‘steps’ in the function course. Reaching a performance-related milestone, e.g., a new tuned parallel version of the code, means additional development effort. Performance milestones can be reached until the maximum-achievable performance of hardware and application is obtained. More details can be found in Wienke et al. [WMSM16] and the master thesis of Miller [Mil16] that I have supervised in 2016.

9. Related Work

Software engineering offers numerous approaches to derive effort estimations for mainstream software development. These are often based on easily quantifiable product metrics that describe a certain type of software complexity. Therefore, I introduce the most common software complexity metrics and discuss their applicability to HPC codes in Sect. 9.1. In Sect. 9.2, I describe the popular software cost estimation model COCOMO II from SE and argue that it is not directly applicable to HPC setups by identifying numerous challenges. Thus, a new methodology for effort estimation in HPC is needed and introduced in the later sections.

9.1. Software Complexity Metrics

Software complexity metrics belong to product metrics and usually score with being easily quantifiable. Exploiting them as independent base metric for effort estimation, they further need to be predictable. An overview of different software complexity metrics can be found in [YZ10]. Instead of an extensive review of available metrics, here, I focus on their usability for HPC codes where a great amount of time is spent for HPC activities. Especially, I investigate the difference in their quantified values between serial code complexity and code complexity of the parallelized and optimized version. For that, I use exemplary a simple software that computes Pi. Given requirements for this (serial) program are: The program must numerically compute Pi and return its approximation. The accuracy of the resulting Pi approximation must be adjustable by an input parameter. The additional requirement of reaching a certain performance is then tackled by the parallelization of that software. One code solution is illustrated in Lst. 9.1 and basis for software complexity comparisons.

9.1.1. Lines of Code

Lines of code (LOC) are the most popular and most straightforward understandable and interpretable metric for software complexity. It comes in different flavors, so that a detailed and working definition is crucial for obtaining consistent results.

Listing 9.1: Simple OpenMP program for the computation of Pi.

```
1 double CalcPi (int n)
2 {
3     // n is number of grid points for numerical integration
4     const double fH = 1.0 / (double) n;
5     double fSum = 0.0;
6     double fX;
7     int i;
8
9     #pragma omp parallel for private(i, fX) reduction(+:fSum)
10    for (i = 0; i < n; ++i)
11    {
12        fX = fH * ((double)i + 0.5);
13        fSum += 4.0 / (1.0 + fX*fX);
14    }
15    return fH * fSum;
16 }
```

Exemplary types are total LOC, modified LOC, added and removed LOC, non-comment LOC or LOC excluding empty lines or certain statements like header inclusions. The software cost model COCOMO II (introduced in Sect. 9.2.1) assumes so-called logical lines of code (LLOC) whose counting rules have been defined with help of the definition checklist of the Software Engineering Institute (SEI) [BAB⁺00a, pp. 77]. LLOC include, for example, executable statements, declarations, compiler directives and expressions terminated by semicolons. Instead, they exclude comments, blank lines or block statements for (sub)program bodies. Nevertheless, counting rules are still ambiguous due to language-dependent characteristics.

As evident, an inherent characteristic of the LOC metric is its dependency on the programming language used in the code, e.g., high-level languages commonly result in fewer LOC than low-level approaches. Furthermore, LOC can vary greatly based on programming style and programmer's efficiency, e.g., using single- vs multi-line if statements, or writing subroutines vs. inlined code. Finally, as the precise quantification of the LOC metric obviously depends on completely-written code, its estimation suffers from great uncertainty.

Pi Example

Investigating the Pi example and counting the total (physical) lines of code in Lst. 9.1, the serial code (omitting line 9) yields 15 LOC, while the OpenMP version (including line 9) delivers 16 LOC.

When adhering to COCOMO's definition of logical lines of code, the serial program accounts for 9 LLOC and the OpenMP program for 10 LLOC. These numbers are based on inclusion of the function definition with corresponding braces (lines 1, 2 and 16) counted as single LLOC, the four declarations in lines 4 to 7 and two executable statements (lines 12 and 13). Furthermore, the expressions in the for-loop are counted as single LLOC while excluding corresponding iteration braces, and the return statement is also counted once. For the parallel code, the compiler directive in line 9 adds one LLOC. The remaining lines are ignored.

Thus, OpenMP parallelization of the Pi code attributes to only one line difference compared to serial code development in both LOC metrics.

Lines of Code in HPC

Factually, the LOC metric is widely used in the context of HPC as evident in numerous publications. Most authors apply LOC for the comparison of effort and complexity of different parallel programming paradigms or library approaches. For example, Steuwer et al. [SKG12] compare the number of code lines written with their SkelCL library for high-level (multi) GPU programming to LOC written with OpenCL and CUDA and conclude that their library reduces LOC on the host side. Instead, Funk et al. [FBHK05] and Hochstein et al. [HCS⁺05] focus on the comparison of OpenMP and MPI. For that, Funk et al. investigate LOC from the NAS Parallel Benchmark Suite, HPCC Suite and different classroom experiments using the *SLOCcount* tool that returns physical LOC. They evaluate these LOC in their relative development time productivity metric (see Sect. 3.2.1). Hochstein et al. also base LOC counts on OpenMP and MPI student codes from classroom assignments. They relate these LOC to development effort logged with diaries and conclude that LOC alone is not a good proxy for effort. An investigation of UPC vs. MPI is conducted by Cantonnet et al. [CYZEG04] and Patel and Gilbert [PG08]. Cantonnet et al. use LOC (excluding comments) and the number of characters as proxy for “manual effort” and compare both programming models with respect to the NAS Parallel Benchmarks, three further kernels and the LOC of the respective sequential code versions. Patel and Gilbert also count non-commented LOC, but base their work on codes from classroom experiments. They focus on statistical hypothesis testing for finding significant differences between LOC of UPC and MPI. In my previous works [WPM⁺11][WSTaM12] that cover two real-world codes, the usability of OpenMP, OpenACC, PGI Accelerator, CUDA and OpenCL is compared by looking at their code expansion (amongst others). There, I work with added and removed physical lines with respect to the serial code version and distinguish between kernel and host code lines. Similarly, the studies of Chamberlain et al. [CDS00] and Christadler et al. [CES12] also compare a wide range of parallel programming models using LOC, but both using NAS or linear algebra/FFT kernels. For evaluation of language expressiveness, Chamberlain et al.

give a rough definition of lines of codes that includes declarations, communication (synchronization, interprocessor data transfers) and computations, but excludes non-essential lines such as initialization, timings and I/O. In contrast, Christadler et al. take LOC from developer diaries in the context of their PRACE project that evaluates performance and productivity of various parallel programming models. They identify unclear LOC definitions as one shortcoming of the project.

Generally, in HPC, the LOC metric is as debatable as in SE including uncertain counting rules, dependencies on the (parallel) programming model and on developer coding styles and skills. Uncertainty in counting definitions even increases in HPC when the focus is actually on the additional complexity introduced by parallelism. Thus, the LOC metric may consider only kernel code lines that encapsulate parallelism, e.g., with OpenMP directives or MPI calls, or also possibly surrounding code. Furthermore, differences in complexity of code lines are neglected which especially affects parallelization activities: For example, LOC assumes the same development cost for a simple line declaring data as for an OpenMP directive entailing thinking time on parallel correctness and efficiency as illustrated in the Pi example. In addition, it does not account for HPC activities outside of source code such as optimizing compiler flags, process binding, debugging and testing. Finally, as in SE, accurate prediction of LOC in early software stages is not feasible.

9.1.2. Function Points

Another popular software complexity metric are function points (FPs) that measure “the (functional) size of software from the user perspective” [BD08]. Function points were introduced by Albrecht in 1979 [Alb79] and the International Function Point Users Group (IFPUG) formalized his method and released corresponding counting rules. The IFPUG defines the unadjusted function points count (UFC) that is based on transactional and data function types. Transactional functions are external inputs (EI), external outputs (EO) and external inquiries (EQ) [BD08, p. 347]: External inputs describe control information that comes from outside the application, whereas EO present data or sent control information to the outside of the application. External inquiries are similar to EO, but do not contain any mathematical formulas and do not create any derived data. Data functions are internal logical files (ILF) and external interface files (EIF). Internal files are meant to maintain and store data within the application, and EIF represent data maintained outside of the application, i.e., by another application. To obtain the UFC, each item is weighted by a “subjective complexity rating” [FB14, p. 352]: low, average, and high. Corresponding weights can be found in Tab. 9.1. The UFC is taken as input for various software estimation tools.

If managers need to directly incorporate further impact factors into their software cost estimation, the IFPUG defines the adjusted function points for that.

Adjusted function points depend on 14 general system characteristics (GSCs), e.g., performance, installation ease, online data entry, or reusability, that get rated from 0 to 5 where 0 means irrelevant, 3 the average and 5 essential [FB14, p. 354]. The value adjustment factor (VAF) is then computed by $VAF = 0.65 + 0.01 \sum_i^{14} GSC_i$ and the adjusted function points by $FP = UFC \times VAF$.

Since function points are based on the user requirements and defined to be independent of the underlying programming model, they are reasonably predictable in early stages of the development process. Moreover, it has been shown [AG83] that they correlate with the number of LOC.

Pi Example

Given the user requirements for the simple Pi example, one EI and EIF can be identified that handle the accuracy of Pi. The computed Pi value is represented by one EO. I assume that corresponding complexity weights are low due to this very simple setup. Thus, the Pi application yields an UFC of 14 (see Tab. 9.1) independent of a serial or parallel realization. The performance requirement comes only into play when computing the VAF. For that, I assume a high rating of 5 for the *performance* GSC (and ignore it for the serial code version). I further set *complex processing* and *reusability* to 4, and *heavily used configuration* and *transaction rate* to 2. The remaining factors are assumed to be irrelevant. From that, I derive for the parallel program $VAF = 0.65 + 0.01 \cdot (5 + 4 + 4 + 2 + 2) = 0.82$ and, thus, as adjusted function points $14 \cdot 0.82 = 11.48$. In comparison, the serial program yields $VAF = 0.77$ and 10.78 adjusted function points.

Interestingly, when converting UFC to LOC given the conversion ratios for the C language of 128 LOC per UFC [BAB⁺00a, p. 6], it comes to 1,792 LOC. This number differs greatly from the computed LLOC number in the previous section, probably due to the small-size example.

Function Points in HPC

Although function points are popular in SE, they have been scarcely applied to parallel software. For example, Post and Kendall [PK04] investigate large-scale nuclear weapons simulation codes at Los Alamos and Lawrence Livermore National Laboratory and compute function points backwards from LOC in C, C++ and Fortran sources. They use function points to estimate a (project-calibrated) time schedule and required team size which are then compared to the amount of actual project efforts and team sizes.

Although function points can be predicted in early stages of development, they cannot directly be mapped to effort measures for HPC purposes since they share

Table 9.1.: Unadjusted function point count of Pi example. Underlined numbers are weights relevant for the example.

Item	Complexity Weighting			Pi example	Result
	Low	Average	High		
External inputs (EI)	<u>3</u>	4	6	1	3
External outputs (EO)	<u>4</u>	5	7	1	4
External inquiries (EQ)	3	4	6	0	0
External files (EIF)	<u>7</u>	10	15	1	7
Internal files (ILF)	5	7	10	0	0
Unadjusted FP count (UFC)					14

several challenges of the LOC metric applied to HPC setups such as ignoring non-functional process requirements. Complexity hidden in parallel programming, tuning or compiler flag optimization is not covered in unadjusted function points and only considered in a subjective weighting factor for adjusted function points. The performance factor can only modify the function point value up to 3 %.

9.1.3. Halstead Complexity Metric

The Halstead Complexity Metric [Hal77] captures size and complexity by splitting a program into a collection of tokens, i.e., operators and operands. Operators are usual arithmetic and logical operators, but also delimiters and pairs of parenthesis. Operands are variables and constants. Then, Halstead defines the program length N as $N = N_1 + N_2$, where N_1 is the number of total occurrences of operators and N_2 the total number of operands. He further defines the vocabulary as $n = n_1 + n_2$ with n_1 the number of unique operators and n_2 the number of unique operands. From that, the volume V of a program is derived that represents the “number of mental comparisons needed to write a program” [FB14, p.345]: $V = N \log_2 n$. The difficulty D and effort E are given as $D = (n_1/2) \cdot (N_2/n_2)$ and $E = V \cdot D$. On the technical side, Halstead also provides a conversion to gain the time T to understand or implement the program. For that, the so-called Stroud number is applied, which denotes the processing rate of the human brain and varies in the range of [5; 20] (default value 18 for software scientists) [Hal77]. Hence, this time is given as $T = E/18$ (in seconds).

While the program length N satisfies the characteristics of a software size measure, the mathematical models of volume V and difficulty D have no connection to the real world [FB14, p.345]. Another drawback captures the estimation capabilities of the Halstead metric: operands and operators are such a fine-grained measure that effort E and time T can only be computed in a post-development

Table 9.2.: Halstead's operators and operands of Pi example.

Operator	#	Operator	#	Operand	#	Operator Operand		
=	4	;	9	fH	3	Total	$N_1 = 38$	$N_2 = 23$
<	1	for	1	fSum	3	Unique	$n_1 = 15$	$n_2 = 9$
+	2	int	1	fX	4	Table 9.3.: Including OpenMP pragma.		
++	1	double	5	i	5			
+=	1	return	1	n	2	Operator Operand		
*	3	const	1	0	2			
/	2			0.5	1	Total	$N_1 = 48$	$N_2 = 26$
()	5			1.0	2	Unique	$n_1 = 22$	$n_2 = 9$
{ }	1			4.0	1			

process. Instead, Halstead suggests a workaround [Hal77, p. 74] that is based on predicting the language level λ and the conceptually unique input and output parameters (operands): n_2^* . From these, other parameter quantifications are likely to be derived in an iterative process (since no closed-form solution exists).

Pi Example

For the application of Halstead's Complexity Metric to my Pi example, I count the operators and operands in Lst. 9.1 ignoring the function definition. The corresponding values for the pure serial version of the Pi example are given in Tab. 9.2. Here, it holds that $N = 61$, $n = 24$, $V = 279.68$, $D = 19.17$ and $E = 5,360.59$, so that $T = 297.81 \text{ s} = 4.96 \text{ min}$.

Next, I examine the whole program in Lst. 9.1 including the OpenMP directive in line 9. I take as operands the variables `i`, `fX` and `fSum`, while I consider as operators `:`, `+`, `,`, `()`, `pragma omp parallel for private reduction` (with results in Tab. 9.3). With that, $N = 74$, $n = 31$, $V = 366.61$, $D = 31.78$, $E = 11,650.07$ and $T = 647.23 \text{ s} = 10.79 \text{ min}$. Thus, it would take roughly 6 minutes more to write the whole OpenMP program than the serial code version. However, with our given focus in the parallel-only part, i.e., just adding the OpenMP directive to the code, it holds that $T = 13.36 \text{ s}$.

Halstead's Metric in HPC

Few works have applied the Halstead complexity metric to parallel programs. Legaux et al. [LLJ14] use Halstead's effort metric to compare a C++ + MPI algorithm to a high-level approach using parallel skeletons hidden in a C++ library both implementing the all nearest smaller values (ANSV) problem. They briefly

define how they interpret operators and operands in C++ programs and especially take (MPI) function calls as operators. Similarly, Coullon et al. [CL13] use Halstead effort to compare an MPI program to a parallel C++ library implementation called SkelGIS solving the heat equation.

While the definition of Halstead's tokens is straightforward for MPI (function calls), it is incomplete for directive-based parallel programming, e.g., using OpenMP. Thus, the OpenMP operand/operator assignments in the Pi example have been freely chosen. In the Pi example, it is also evident that the interpretation of parallel-only effort is ambiguous yielding very different results. Especially, assuming the addition of statements for parallelism, the time result is very low (see $T = 13$ s for the OpenMP directive in the example). Furthermore, this metric does not consider any effort spent outside of source code as, e.g., compiler flag optimization. The usability of Halstead's time metric in HPC could not be verified by any publication. Finally, due to the fine-granular definition of tokens, it is hardly possible to predict the numbers in advance. Therefore, complicated workarounds must be eventually applied that have not been studied much in literature.

9.1.4. Cyclomatic Complexity Metric

McCabe introduced his cyclomatic complexity metric in 1976 [McC76]. It follows a graph-theoretic concept and depends on the decision structure, i.e., the control flow, of a program. The cyclomatic complexity measures the number of basic paths that can be combined to create every possible path through a program by $V(G) = e - n + 2p$ where G represents the program's control flow graph with e edges and n nodes. The variable p describes the number of connected components where a component can be, e.g., a module or function. As simplification for single-component graphs, McCabe's cyclomatic measure can be computed by counting the number of predicates (or rather conditional statements) and adding one.

McCabe intentionally invented his metric to define an upper limit on complexity for developing modules and argued that larger modules are more difficult to test and maintain. Thus, the (cyclomatic) complexity of a program is also likely to correlate with the effort required to develop and test a program. Nevertheless, this metric has been actually introduced as measure and, hence, its granularity is still very fine to get accurate predictions of a program's complexity. When focusing on the estimation of the number of components p , a reasonable value is more likely. However, it still does not provide an actual effort value since its correlation to development and testing is unknown.

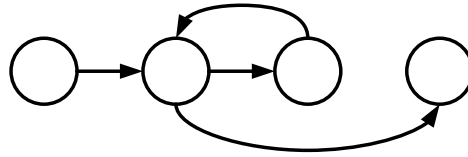


Figure 9.1.: Control flow graph for McCabe's cyclomatic complexity of Pi example.

Pi Example

Measuring cyclomatic complexity of the Pi example results in a simple control flow graph with $e = 4$ edges and $n = 4$ nodes (compare Fig. 9.1). Because the Pi example is implemented in a single function without connectivity to other modules, it holds that $p = 1$ and, hence, $V(g) = 4 - 4 + 2 \cdot 1 = 2$. The simplified method delivers the same result: One condition within the for loop yields $V(g) = 1 + 1 = 2$.

Since the cyclomatic complexity focuses on the program complexity expressed in source code instead of the execution complexity, the parallel version of Pi using OpenMP directives equates to the same complexity number as the serial version.

Cyclomatic Complexity in HPC

Research on the application of cyclomatic complexity to codes written with parallel programming languages has rarely been undertaken, only VanderWiel et al. published some works [VNL96][VNL97][VNL98] in the late 90s, where [VNL96] and [VNL98] just give some more details on the work conducted in [VNL97]. They use cyclomatic complexity (and non-commented source code statements) to quantify the relative effort and ease of use of different parallel programming models in terms of message passing, shared memory parallelization and High-Performance Fortran. For that, they focus on user-level code within source files of six test programs and ignore (parallel) execution control flows. Results are presented normalized to the equivalent serial code values and are in line with the authors' experiences. Nevertheless, they state that the validity of extending complexity measures to parallel programs must be further investigated.

The works of VanderWiel et al. also show that message passing adds only high complexity to irregularly or specially-conditioned communicating applications. Thus, if control flows do not change by adding parallelism or through tuning activities, such as compiler flag optimization, thread binding, or extension by OpenMP directives (see Pi example), cyclomatic complexity does not cover well required efforts for parallel development. In addition, its predictability capabilities is troublesome analogously to the estimation of serial-program control flows.

9.2. Software Cost Estimation Techniques

In the planning phase of traditional software project management, the estimation of software cost is one essential part. From that, estimations for schedule, budgeting and risk can be derived. Thus, software cost estimation techniques have been extensively studied in SE. Estimation approaches cover expert judgment, analogy-based models, learning-oriented models, dynamics-based techniques and parametric or algorithmic models. A brief overview of some estimation methods with their strengths and weaknesses can be found in [SBL12]. More details and more estimation techniques are surveyed by Boehm et al. [BAC00]. Especially in parametric or algorithmic models, software complexity metrics as introduced in Sect. 9.1 can serve as independent base metric used to estimate effort, software cost and schedule. In the following, I focus on the parametric COCOMO II model with LOC as main base metric since I modify the approach taken from COCOMO II for my effort estimation method covered in Sect. 10.1.

9.2.1. COCOMO II

While many proprietary models exist, the *Constructive Cost Model (COCOMO)* is openly accessible and with its parametric behavior easy to understand and analyze. It is a “popular approach” [FB14, p. 110] that is taught to several hundreds estimators by Steve McConnell [McC06, p. 47], the Chief Software Engineer at Construx Software.

COCOMO has been originally introduced by Boehm [Boe81] in 1981 and released as *COCOMO II* including updates to modern software life-cycles and programming languages [BCH⁺95][BAB⁺00b][BAB⁺00a]. Here, I cover COCOMO II and I use the terms COCOMO II, COCOMO II.2000 and COCOMO interchangeably in the remainder of this work. COCOMO II models effort (in person-months) as function of software size, i.e., thousand lines of codes (KLOC) or function points. Its parameters were set up and calibrated using multiple regression analysis on survey results from 161 large projects [BCH⁺95]. They represent cost drivers of the software project and software team and are distinguished between so-called *effort multipliers (EMs)* and *scale factors (SFs)* (compare Tab. 9.4). EM describe “characteristics of the software development”, whereas SF account for “the relative economies or diseconomies of scale encountered for software projects of different sizes” [BAB⁺00a]. A short description of all EMs and SFs can be found in Tab. A.7 and A.8. Cost drivers are assessed by users on Likert-type rating scales from *very-low* to *extra-high* and then translated to numerical values where their *nominal* values are set to 1. Some exemplary Likert-type scales are illustrated in Tab. 9.5.

COCOMO originally based on the Waterfall software life-cycle and has been updated to additionally accommodate spiral processes covering phases such as

Table 9.4.: Scale Factors (SFs) and Effort Multipliers (EMs) used in COCOMO II. Significance values are computed by dividing the largest by the smallest value defined by COCOMO II's rating scale [McC06]. SF values depend on projects with 100 KLOC.

Cost driver	SF	EM	Signif.
Product Complexity (CPLX)		×	2.38
Analyst Capability (ACAP)		×	2.00
Programmer Capability (PCAP)		×	1.76
Execution Time Constraint (TIME)		×	1.63
Personnel Continuity (PCON)		×	1.59
Multisite Development (SITE)		×	1.56
Required Software Reliability (RELY)		×	1.54
Documentation Match to Life-Cycle Needs (DOCU)		×	1.52
Applications Experience (APEX)		×	1.51
Use of Software Tools (TOOL)		×	1.50
Platform Volatility (PVOL)		×	1.49
Main Storage Constraint (STOR)		×	1.46
Process Maturity (PMAT)	×		1.43
Language & Tool Experience (LTEX)		×	1.43
Required Development Schedule (SCED)		×	1.43
Data Base Size (DATA)		×	1.42
Platform Experience (PLEX)		×	1.40
Architecture/ Risk Resolution (RESL)	×		1.38
Precedentedness (PREC)	×		1.33
Developed for Reusability (RUSE)		×	1.31
Team Cohesion (TEAM)	×		1.29
Development Flexibility (FLEX)	×		1.26

inception, elaboration, construction, and transition [BAB⁺00a]. Depending on the information that is available in that phase, the COCOMO II model actually provides three approaches: Information for the first approach can be derived from prototypes or initial specifications (Early Prototyping stage). As the life-cycle proceeds, the characteristics of the software are better tangible (Early Design stage) and detailed when the “definition of the system and software architecture itself” [Boe96] is completed (Post-Architecture stage). This life-cycle process also affects uncertainty in estimation results and is further described in Sect. 9.2.2. In the Early Prototyping stage, COCOMO estimations are based on so-called Object Points that are counted and weighted with complexity ratings from *simple* to *difficult*. COCOMO Object Points are not specialized to object-oriented concerns, but focus on screens, reports and third-generation language modules, and hence, on

Table 9.5.: Exemplary Likert-type rating scales used in COCOMO II [BAB⁺00a].

Scale Factors	Very Low	Low	Nominal	High	Very High	Extra High
Architecture / risk resolution	none	little	some	generally	mostly	full
Documentation Match to Life-Cycle Needs	Many life-cycle needs uncovered	Some life-cycle needs uncovered	Right-sized to life-cycle needs	Excessive for life-cycle needs	Very excessive for life-cycle needs	

software built with GUI-builder tools. COCOMO’s Early Design Model works with unadjusted function points (compare Sect. 9.1.2) that are transferred to SLOC with tabulated conversion factors. It uses the same basic functional formula as defined in the Post-Architecture stage with a reduced number of EM, i.e., $h = 7$. The more detailed Post-Architecture Model focuses on the development and maintenance of a project. Its independent base metric of software *SIZE* must be denoted in KSLOC or as unadjusted function points that are translated to KLOC. Counting rules on the interpretation of LOC are specified in COCOMO’s Model Definition Manual [BAB⁺00a] and have been exemplary shown in Sect. 9.1.1. Furthermore, the Post-Architecture Model defines 17 EMs ($h = 17$) and 5 SFs, so that:

$$\text{effort [person-months]} = A \cdot M \cdot \text{SIZE}^F \quad (9.1)$$

$$M = \prod_{i=1}^h \text{EM}_i \quad (9.2)$$

$$F = B + 0.01 \cdot \sum_{j=1}^5 \text{SF}_j \quad (9.3)$$

where A and B are calibration constants that should be adapted to the local development environment to get higher accuracy (defaults are $A = 2.94$ and $B = 0.91$) [BAB⁺00a].

The model and *SIZE* metric in Equ. (9.1) is intended for effort estimations of new developments, i.e., writing software from scratch. If this newly-developed software can reuse existing code modules from other software packages, e.g., from libraries or previously-written codes, these existing SLOC contribute to the software size as well. Therefore, COCOMO also provides an additional *Reuse Model* that accounts for reused, adapted and automatically-translated code in so-called equivalent source lines of code (ESLOC) including additional effort for familiarization with the existing code, for modification of the code for inclusion in the new software or automatic translation.

Since COCOMO II approaches have been examined carefully in research, some problems were reported but also further improvements were suggested. Boehm et al. [CBS99] find counter-intuitive results for some parameters evoked by the regression analysis of COCOMO. Some given reasons are insufficient data points for the regression analysis, outliers or correlation of parameters. Thus, gathering

numerous data points is important. Calibration techniques such as constraint multiple regression [NSB08] or an additional Bayesian approach [CBS99] that includes expert judgment into the model can further improve the accuracy of the model. Another chance to increase accuracy is a local calibration to own institutional data, e.g, adapting constants A and B in Equ. (9.1) and (9.3).

COCOMO II in HPC

While the COCOMO model has been widely examined in software engineering research, only few publications touch upon COCOMO II in the HPC domain. In [Kep04a], Kepner discusses some mainstream software engineering results with respect to HPC. There, he briefly mentions the existence of COCOMO as example that effort in SE can be related to SLOC and concludes that SLOC are not a good measure to evaluate programmers. Similarly briefly, Snir and Bader [SB04] reference COCOMO as one software estimation model that is based on various program metrics similar to their formula of time to solution as function of system and problem parameters (also compare Sect. 3.3.1). In my previous work [WIaMM15], I started elaborating on the usage of COCOMO II for HPC. I postulate the hypothesis and idea that, in HPC, COCOMO's LOC should not represent the number of LOC of newly-developed code, but describe the parallelizable (kernel) code of the application. Further, I simplify the linear factor given as effort multipliers to a single calibration parameter. The calibration parameter assumes that some base effort has already been spent for parallelizing/ tuning a single kernel and that it can be scaled to (potential) multiple kernels or large codes. For simplicity, I also use generally low ratings for all scale factors so that the diseconomy of scale slightly affects the effort result. The resulting relationship of increasing code parts to parallelize, and effort is illustrated in a real-world HPC setup.

In a next step, Miller and I have investigated the applicability of COCOMO II to HPC projects in a comprehensive study described in [NMW⁺16] and [MWSL⁺17]: We looked into two case studies with a total of 28 data sets. The first case study is based on classroom experiments conducted in the context of student software labs at RWTH Aachen University from 2013 to 2016 [WCM16]. There, students had to parallelize a serial Conjugate Gradient (CG) Solver with OpenMP, OpenACC and CUDA. The full study setup is described in Sect. A.1.3. The second project covers HPC activities conducted for the real-world aeroacoustics simulation code *ZFS*. Here, the Discontinuous Galerkin solver of *ZFS* was parallelized with OpenACC and highly tuned for GPUs. This case study is covered in-depth in Part D. After completion of both projects, we applied COCOMO's Post-Architecture Model to these projects by gathering all information required to rate the 17 EMs and 5 SFs and interpreting these cost drivers in the context of HPC. We expressed COCOMO's *SIZE* metric as difference of SLOC of the base code version and the parallel/tuned version. We also considered lines of existing code that needed to be

rewritten to ensure correct execution on the GPU by applying parts of COCOMO's Reuse Model. Finally, we compared corresponding effort estimations to actual efforts that were tracked by the developers using diaries or logging tools (Sect. 11.2) and applied an uncertainty and sensitivity analysis to account for possible errors in our assumptions. We found that the development efforts of the software labs and the basic parallelization of *ZFS* were mostly overestimated by COCOMO. This is surprising given the expectation that parallelization requires more time than mainstream software development. Possible reasons were identified in the small number of LOC and the uncertainty and subjectivity in cost driver evaluation. A better point estimate was achieved for the aggregation of all HPC activities of *ZFS* (basic parallelization and different tuning steps), however, it still suffered from great variances when assuming inaccuracies in the input parameters. Furthermore, this seemingly increased accuracy was rather a result due to compensation of the high efforts needed for tuning. Consequently, COCOMO underestimated effort for the applied tuning activities.

From our experiences gathered in the case studies, we have derived the following challenges in the applicability of the COCOMO model to HPC projects mostly described in [WMSM16][MWSL⁺17][Mil16].

Lines of Code as Base Metric Since COCOMO is based on lines of code as independent size metric, it comes with all the drawbacks from LOC discussed in Sect. 9.1.1. This includes challenges in predictability and the need for accounting extra effort for lines (or compiler flags) that target parallelization and tuning.

Counting Rules for Lines of Code COCOMO II further assumes development of new software and defines counting rules for inclusion and exclusion of statements. Contrarily, HPC activities often do not start from scratch, but on existing domain-specific code (compare Sect. 8.1.1 for more details). In this case, it is unclear which lines to count: lines that account for parallelization and tuning such as OpenMP directives or MPI calls, the whole kernel code lines or all modified lines. The latter might differ from kernel code lines, e.g., due to data structures that must be optimized or restructured when targeting different architectures. In addition, we noticed in our case study that in an iterative parallelization process, some lines were adjusted multiple times. A simple line count cannot accommodate that.

Model Parametrization We also ascribe COCOMO's weak HPC fit to the kind of historical data that was originally used to parameterize and calibrate the model. This data is based on numerous large projects with (several) hundred thousand lines of code. While the domain-specific base code in HPC projects might contain multiple hundred KLOC, the amount of parallelization statements or kernel code

lines are rather in the order of hundred to thousand LOC and, thus, way smaller. Furthermore, the COCOMO data sets come from projects with big developer and analyst teams. This is rather uncommon in the environment that we have assumed where mainly few HPC experts are responsible for parallelization and tuning.

Interpretation of Parameters Besides the actual parameterization and calibration of the COCOMO model, most parameters relate to traditional software development not having performance-oriented setups in mind. This is especially evident when available (survey) ratings must be interpreted in the context of HPC. For example, the DATA parameter relies to databases, the product complexity (CPLX) also includes data management operations or user interface management operations, and the architecture-risk resolution (RESL) asks for risk management plans in schedule, budget and milestones with support by risk tools. Although COCOMO provides the TIME cost driver that represents execution time constraints, it only covers performance in four percentage ranges of available execution time: First, the baseline for the percentage estimation is not given. In an HPC context, roofline models for single compute nodes or LogP models for network-related execution could be applied. However, this definition has not been investigated so far and percentage values might be difficult to obtain. Second, the term “available execution time” is not well defined in HPC where hundreds of compute nodes are leveraged for execution runs. Third, the four-point rating scale provides only very limited coverage of a complex performance tuning process. At least, the translation of its rating scale to numerical values equates very roughly to exponential growth when encountering higher constraints.

Subjective Rating The challenge of interpreting COCOMO cost drivers in the context of HPC is further increased through its various subjective rating scales. Subjective rating scales are prone towards bias and are critically mentioned by McConnell [McC06, p. 47]. Exemplary subjective rating scales for PREC, RESL and DOCU are illustrated in Tab. 9.5. Moreover, the corresponding reference groups (for nominal ratings) are unknown. Similarly, reference descriptions are often unclear causing a non-objective rating with potential errors. For example, COCOMO covers experiences of developers (AEXP, LTEX, PEXP) in ratings given by the number of months or years the developers had “equivalent level of experience” with the same type of application, tool, language or platform. However, the meaning of “equivalent level of experiences” is not well defined. Additionally, time periods (as rating scales) are probably not able to fully accommodate the complex characteristics of application, language or tool experiences.

Interpretation of Reuse Model As mentioned above, COCOMO also offers a reuse model that is actually intended to account for effort needed to take code from

other applications and incorporate it into the new software. Given the HPC landscape where (domain) base code exists and is extended with parallelism and performance optimizations, the re-interpretation of the reuse model in this context seems fair. In [Mil16][MWSL⁺17], Miller proposes to use COCOMO's re-engineering approach for automatically-translated code [BAB⁺00a, p. 13]. For that, all modified SLOC within the existing base code are counted as so-called adapted SLOC (ASLOC). Then, the percentage of automatically-translated code is interpreted as interactive development yielding a factor of 50 % obtained from COCOMO tables, and the required rate of translated statements per month is set to 1 since no automatic translation takes place. COCOMO's reuse model [BAB⁺00a, p. 9,71] further suggests a nonlinear extension of ASLOC by estimating the so-called equivalent SLOC (ESLOC) that account for the understandability of existing software (SU) or the programmer's relative unfamiliarity with the code (UNFM). Corresponding weights (based on subjective rating scales) are added to an adjustment factor that covers, e.g., the percentage of code that is adapted for integration into new software. In future work, I will reinterpret this extended reuse approach for HPC setups to incorporate extra efforts when dealing with foreign code. Here, I suggest to translate COCOMO's percentage of modified code to the existing domain code that is parallelized or tuned. Furthermore, the SU factor can be assumed to depend on the complexity of the domain code structure and clarity, while the UNFM factor might be interpreted in terms of HPC experts that first have to understand the basics of the domain code before any HPC activities can be applied.

Testing COCOMO does not directly capture effort for testing software, but only encapsulates some minor test components into effort multipliers such as DATA. Additionally, Aranha and Borba [AB07] state that COCOMO is not able to estimate effort to execute or automate test data sets. However, testing takes a huge amount of time in HPC projects (as discussed in Sect. 8.1.1): Kepner [Kep04a] illustrates a share of 34 % to 50 % for testing, while Miller [Mil16] spent 56 % for debugging and testing in his tuning effort of *ZFS* (Part D). Thus, COCOMO might not account for this extra testing effort. In future, I will further investigate the impact of testing effort by means of HPC effort logs of students and developers.

Overall, the COCOMO II model is not directly applicable for effort estimations in HPC environments due to severe differences in mainstream software development and development in HPC, the challenges given above and its focus on relating all parameters to source lines of code.

9.2.2. Uncertainty and Accuracy

As seen in Sect. 4.6, productivity estimation is subject to uncertainty if variability in the input parameters occurs. The same holds for software cost estimations

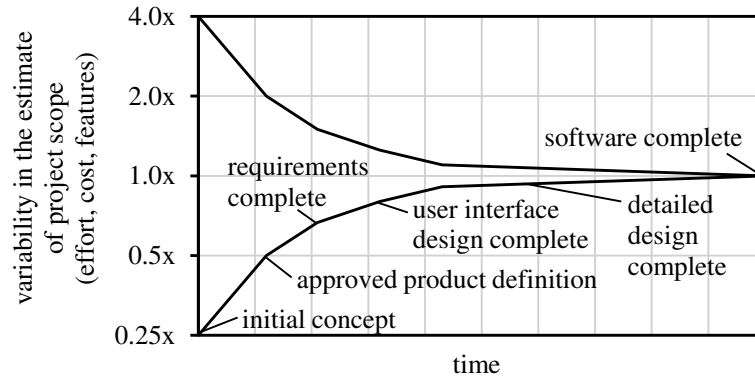


Figure 9.2.: Cone of Uncertainty taken from [McC06, p. 37].

and, thus, is one reason for the variability in productivity. In SE, the predictable amount of uncertainty over time is denoted by the *Cone of Uncertainty* that suggests a degree of error found in estimates. This behavior taken from software engineering must also be accounted and applied for setting up an effort estimation model that suits HPC environments.

Cone of Uncertainty

The uncertainty in estimates highly depends on the project stage. Early in a software project, most details on software requirements, solution possibilities and staffing are unknown. Thus, inaccuracies in estimates can be encountered in the order of a factor of 4 on the high or low side [McC06] for a skilled estimator, i.e, in the best case. Over time, the variability in uncertainty diminishes when decisions get implemented. This behavior is depicted in Fig. 9.2 for a sequential software development process. For an iterative process, each iteration is subject to a small cones of uncertainty.

Besides noting errors in the project itself, McConnell [McC06] also states several reasons for errors occurring in common estimation practices. For example, he explains that certain activities are often omitted like ramp-up time for new team members, creation of test data or performance tuning. Furthermore, developers and managers tend to adhere to (unfounded) optimism when predicting required effort. Research has found that estimates of developers often comprise an optimism factor of 20 % to 30 %. He further gives as reason subjectivity, bias or unfamiliarization with a certain technological area.

Estimations of development effort in the context of HPC activities suffer from the same errors and sources of uncertainty. Given the cone of uncertainty, it is obvious that point estimates done before HPC activities start will be subject of variability. Nevertheless, an alternative (more accurate) approach does not exist

and, thus, my goal is making a reasonable first estimate on development effort to include it into productivity predictions of HPC centers. This also emphasizes that an uncertainty analysis of the productivity metric is an important step to judge the reliability of results. Finally, when relating COCOMO's different models to the cone of uncertainty, it gets clearer that its Post-Architecture model is not usable for early predictions, however, it can still serve as foundation of further investigations in HPC.

Measures of Estimation Accuracy

Since uncertainties lead to errors in estimates, it is recommended to measure the error quantities of predicted values during the development process and validate the quality of an estimation model with respect to actual encountered efforts.

In software engineering, estimation accuracy is typically measured as magnitude of relative errors (MRE) and mean (or median) magnitude of relative error (MMRE) over $i = 1, \dots, n$ observations:

$$\text{MRE}_i = \left| \frac{\text{effort}_{\text{actual},i} - \text{effort}_{\text{estimated},i}}{\text{effort}_{\text{actual},i}} \right| \quad (9.4)$$

$$\text{MMRE} = \frac{1}{n} \sum_{i=1}^n \text{MRE}_i. \quad (9.5)$$

With a small MMRE, the software cost model reasonably predicts (on average) the actual effort, however, outliers can still occur. Conte et al. [CDS85] state that most researchers accept an $\text{MMRE} \leq 0.2$. With respect to possible outliers, the quality of model-predicted efforts is further measured as:

$$\text{PRED}(l) = \frac{o}{n} \quad (9.6)$$

where l is the prediction level (or quality) and o the number of observations where $\text{MRE} \leq l$. For example, $\text{PRED}(0.25) = 0.75$ means that 75 % of the observed effort estimates lie within an estimation error of $\pm 25\%$. Conte et al. [CDS85] assume an effort estimation model can be accepted at $\text{PRED}(0.25) \geq 0.75$.

If a total of n data points are available, the accuracy of the estimation model is usually not only investigated by the complete set of data points, but also by a K-fold cross validation [NSB08].

10. Methodology of Development Effort Estimation in HPC

Software complexity metrics and software cost models, like COCOMO II, come with numerous drawbacks when applied to HPC as discussed in Sect. 9. Thus, an application to HPC is not straightforward. The main reason is that HPC requirements differ from mainstream software development. This difference was also recognized by Snir and Bader [SB04] who concluded in their work that “the development process for high performance scientific codes is fairly different from the development processes that have been traditionally studied in software engineering”. In HPC, the development and use of parallel software, as well as the inherent strong focus on performance require great efforts needed for thinking about (correct) parallelism, analyzing performance, tuning time-consuming code portions and parallel debugging. Therefore, I propose a new methodology to estimate development effort in the area of HPC.

In Sect. 10.1, I give an overview of the methodology whose concept is mainly based on a performance life-cycle as introduced in Sect. 10.2. Section 10.3 shows my approach for the identification of impact factors on effort in HPC, as well, as early results from ranking surveys. Focusing on the key drivers of effort, I apply the concept of knowledge surveys to quantify the impact of pre-knowledge in Sect. 10.4. Ideas on how to quantify the impact of the parallel programming model and numerical algorithm are discussed in Sect. 10.5.

The work presented in this chapter was initiated during collaborative work with the Lawrence Livermore National Laboratory (LLNL), CA, USA. In this context, I started investigating impact factors on development effort and identified two interesting factors, i.e., the parallel programming model and kind of numerical algorithm, and their potential quantification by means of pattern-based categorization of parallel programs. I presented these ideas in a poster [WCMS15] at the International Conference for High Performance Computing, Networking, Storage and Analysis (SC15), discussed the variety of HPC impact factors with the audience and distributed questionnaires to them to get more insight into their development processes and views. My continuous work on development effort estimation in HPC has been published at SC16 [WMSM16] and serves as main basis for the work in this chapter by describing the methodology for HPC effort estimation and presenting first results. Miller contributed to results covered in this section by

implementing the *EffortLog* tool during his student worker job (and its ongoing further development) and by analysis of performance-effort relationships of *ZFS* and student data that I have previously collected in software labs [WCM16].

Furthermore, results presented here are based on empirical data gathered from student software labs (see Sect. A.1.3), GPU hackathons and surveys distributed to various researchers in HPC. In these GPU hackathons, researchers from Europe are mentored over five days by vendors and scientific staff that are proficient in GPU programming with the goal to port or tune brought real-world codes for GPUs. To participate, researchers have to apply in advance with project and goal description, bring at least three developers to the hackathon event and, in return, get two personal mentors for the whole week [Bru17]. During my mentoring job at GPU hackathons in Dresden [TUD16] and Lugano [Swi16] in 2016, I motivated participants to fill in surveys and collect effort logs.

10.1. Methodology Step by Step

My methodology for development effort estimation in HPC introduces an approach to overcome current gaps between estimation capabilities in SE and HPC and is built upon empirical studies. It follows the rudimentary concept of COCOMO II and adapts and extends it for the challenges encountered in HPC.

The core idea of my methodology is to move away from software size as base metric and focus on performance instead. Since most previous software cost models follow traditional software life-cycles, I establish a so-called *performance life-cycle (PLC)* as an appropriate counterpart in HPC. This performance life-cycle defines the relationship between performance and software development effort that is needed to achieve the given performance. However, the amount of development time that needs to be spent further depends on impact factors such as programmer skills, the parallel programming model or used numerical algorithm. These are represented by additional parameters in the effort-performance function. The respective assignment of numerical values for various data sets is specified by my methodology and described in the following. A regression analysis (and possible further improvements mentioned in Sect. 9.2.1) across all data sets can then be applied for model calibration and verified by techniques covered in Sect. 9.2.2.

Consequently, the contribution of impact factors to development effort is investigated. For that, I provide a series of surveys asking developers to rank and name factors important to them. With a statistical analysis, significant differences in ranks can be shown or factors can be combined or eliminated if they do not have sufficient impact on their own. The top entries in the resulting ranking list — also called *key drivers* — are (first) evaluated in detail and included into the performance life-cycle. Moving from ordinal ranks to numerical values, subjective ratings

Table 10.1.: Overview of previous studies on performance life-cycles. P = professional, S = student, EF = survey on effort impact factors, KS = knowledge survey with (pre-post) replies, DD = developer diary (effort-performance pairs), eDD = electronic DD, sDD = summary of DD.

Event	Type	Year	Experiments	#potential teams/people	team replies
Hackathons	P	2015 [Law15]	(EF, DD)	> 20	(8,3)
	P	2016 [TUD16]	(KS,EF,(e)DD)	18	(6-4,6,2)
	P	2016 [Swi16]	(KS,EF,eDD)	18	(3-0,3,0)
Poster survey	P	2015 [WCMS15]	(sDD, EF)	> 10	(0,4)
Software labs	S	2013 [WCM16]	(DD)	8	(8)
	S	2014 [WCM16]	(DD)	6	(6)
	S	2015 [WCM16]	(DD)	7	(7)
	S	2016 [WCM16]	(KS,EF,eDD)	6	(7-2,5,6)
Programming competitions	S	2012/13 [ITC13]	(DD)	> 350	(1)
	S	2015/16 [Cha16]	(EF,sDD)	> 350	(19,10)
Colleagues	P	2016	(EF)	14	(7)
Bachelor theses	S	2014 [Sur14]	(DD)	1	(1)
	S	2015 [Nic15]	(DD)	1	(1)
Student worker	S	2015	(DD)	1	(1)

are avoided as much as possible. For instance, the developer's pre-knowledge is not ranked by years of experience, but knowledge surveys are used.

Moreover, data sets that are necessary to instantiate the PLC model and the (statistical) analysis are collected. These incorporate effort-performance pairs tracked continuously during development, ranking of impact factors on effort and filled knowledge surveys. Gathering data sets is a very challenging task since it means overhead for developers. To ease the collection of effort-performance pairs and increase the accuracy of time logs, Miller and I developed the *EffortLog* tool that is described in Sect. 11.2. Despite the difficulty of getting data sets, I accomplished to gain reasonable feedback over the years on various events which serves as basis for proof of concepts for this methodology. Table 10.1 captures the events and experiences from data collection.

Finally, to reduce bias and increase statistical meaningfulness, I envision a community approach where developers from different institutions and countries collectively gather data sets. To contribute to these efforts and to help jump start this for HPC, I provide survey data and tools publicly available on our webpages [Wie16a].

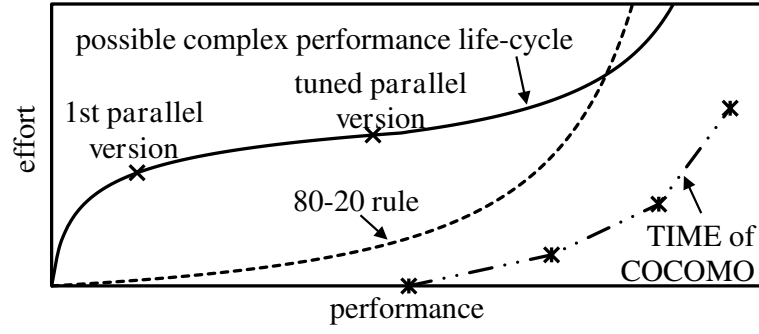


Figure 10.1.: Different performance life-cycle approaches.

10.2. Effort-Performance Relationship

The relationship between performance and effort that needs to be spent on parallelizing and tuning an application to achieve the performance plays a major role in development effort estimation.

10.2.1. Performance Life-Cycle

I introduce the so-called *performance life-cycle* that describes this relationship. Thus, for the estimation of effort in HPC, a performance target must be specified first. Effort is then expressed as function of performance:

$$\text{effort} = S \cdot f(\text{performance})^R + Q. \quad (10.1)$$

Function $f(\text{performance})$ could follow an effort-performance relationship that is as easy as the 80-20 rule, however, the experience shows that this rule does not always suffice. Instead, more complex relationships commonly arise in HPC, of which examples are illustrated in Fig. 8.1 and 10.1.

As in a software life-cycle, the performance life-cycle spans certain performance-related milestones depending on the code starting point (e.g., from scratch or from a working serial code version). For example, developers could start from a working serial code version, create a working parallel version and then achieve multiple times the milestone of a tuned parallel code version when targeting different optimization aspects, e.g., NUMA-affine data access or vectorization. Table 10.2 lists these most important milestones that are encountered in HPC activities and a sample workflow applying OpenMP to a CG Solver is described by Schmidl et al. [SIT⁺14]. Correspondingly, the function f in Equ. (10.1) commonly depends on these milestones as exemplified in Fig. 8.1 and 10.1.

The parameters S , R and Q in Equ. (10.1) represent different impact factors on effort in HPC. As a starting point, I assume a similar composition of cost drivers

Table 10.2.: Possible milestones in a performance life-cycle.

Milestone	Explanation
Working serial version	Serially-running version of the code that was tested for correctness, but not tuned for performance, e.g., 1st correct code implementation
Tuned serial version	Serially-running version of the code that was tested for correctness and tuned for performance
Working parallel version	In-parallel running version of the code that was tested for correctness, but not highly-tuned for performance, e.g. 1st correct parallel version of your code
Tuned parallel version	In-parallel running version of the code that was tested for correctness and tuned for performance

as given by the COCOMO II model, i.e., $S = s_0 \cdot \prod_{i=1}^n s_i$ and $R = r_0 + \sum_{j=1}^m r_j$. To account for one-time efforts, the factor Q is added with $Q = \sum_{k=1}^g q_k$. Nevertheless, the composition is subject to change if analysis suggests another relationship.

10.2.2. Related Work

For the coverage of the dependency between effort and performance, the COCOMO model introduces the TIME effort multiplier. Its rating scale items are given by performance percentages, e.g., “70 % use of available execution time”. When the COCOMO model translates these ratings to numerical values, they show slight exponential growth (compare Fig. 10.1). Drawbacks and challenges of this approach are discussed in Sect. 9.2.1.

Snir and Bader [SB04] cover effort-performance relationships based on their utility theory that uses time to solution (see Sect. 3.3.1). They model execution time as function of development time relying on the 80-20 rule. Due to the corresponding nearly L-shaped curve having two asymptotes, they simplify execution time t to

$$t = \begin{cases} \infty & \text{if } T_{DE} < T_{DE_{\min}} \\ t_{\min} & \text{if } T_{DE} \geq T_{DE_{\min}} \end{cases}$$

with T_{DE} the development time. Furthermore, they conclude that “execution time is strongly affected by investments in program development, and a significant fraction of program development is devoted to performance tuning” so that these two components should not be studied independently of each other.

10.2.3. Statistical Methods

For statistical analysis, I follow the successful approach of the COCOMO II setup using regression as main method. Improvements on initial parametrization can also follow the methods described in Sect. 9.2.1.

10.2.4. Proof of Concept

The common practice in HPC is the focus and measurement of performance of programs. Occasionally, effort spent to parallelize, port or tune these programs is approximated at the end of the HPC project so that a single effort-performance pair is captured. However, this single data set does not cast light on the progressive shape of the effort-performance relationship. Therefore, I motivate the tracking of effort-performance pairs continuously over (development) time using developer diaries and electronic tools (see Sect. 11.2) that record both at certain performance-related milestones. With these two techniques, I accomplished to collect several data sets from student software labs [WCM16] (compare study setup in Appendix A.1.3). While student data is said not to be representative for ‘real’ HPC development [SS10], students are still the best chance to thoroughly test my methodology since they can get easily awarded for participation and generate a reasonable number of data sets. If these student tests prove the concept, the methodology can and should be applied to real-world HPC projects with professional developers. However, today, it is still challenging to motivate HPC professionals to participate and I mainly succeed if they are personally interested in the topic (compare participation results in Tab. 10.1).

The analysis of the effort-performance relationship from my collected student data sets was conducted by Miller as part of his Master thesis [Mil16] with my supervision. For that, we applied the following rules: (a) Development time is multiplied by the number of developers in a student team (here usually two) if all are working simultaneously. (b) Effort denoted for familiarization with the assignment/ numerical algorithm is evenly split across all three programming model approaches. (c) For HPC activities that caused a slowdown in performance, the previously-reached performance was taken. (d) Student submissions that delivered incorrect results are excluded from analysis. (e) The final performance number for each student assignment is verified or overwritten by measurements of the supervisors using the students’ submitted codes.

Our analysis excluded student teams that submitted incorrect solutions, or denoted less than five effort-performance pairs to gain a valid and reasonable progressive function shape. We also normalized the performance data with respect to the runtime of the highly-tuned reference version. The development effort was

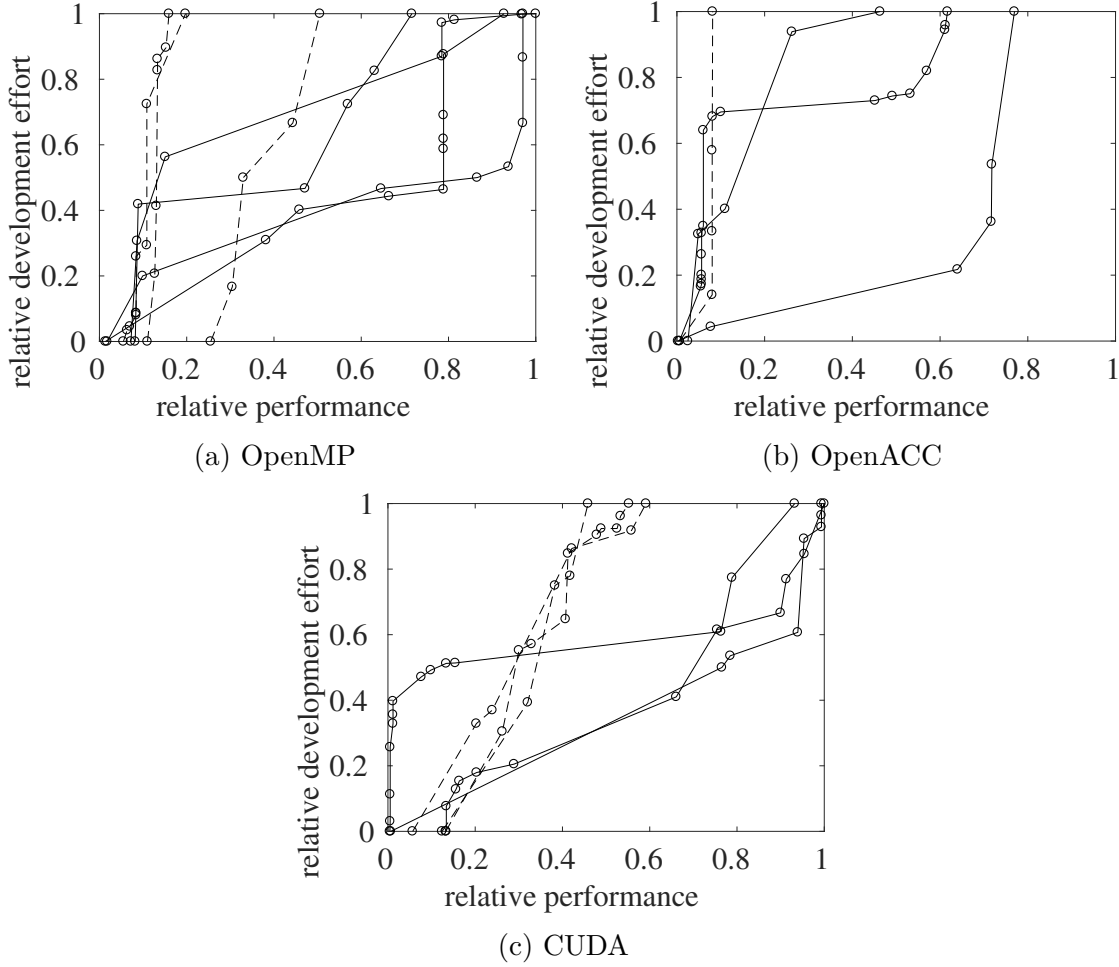


Figure 10.2.: Effort-performance relationships collected from student software labs with different parallel programming models.

normalized within each student team (and programming model) taking the highest effort spent as reference. The corresponding results are shown in Fig. 10.2 distinguished by the used parallel programming model (OpenMP, OpenACC or CUDA). Each curve represents the performance life-cycle for one student team.

Two main distinctions are obvious: Effort-performance relationships with approximately linear function graph (dashed lines) and step-wise function shapes (pure lines) that reach the tuning milestone of parallel code several times. Student teams yielding linear function shapes did not achieve more than 60% of the best-effort performance from any other student team in that year. Most step-wise function profiles show a major performance leap around the mid-point of the project, i.e., 30% to 70% of relative effort. A performance life-cycle that mirrors approximately the 80-20 rule is only followed by a single student team (using OpenACC). Furthermore, most life-cycles reveal two distinct stages. An

initial phase that covers the initialization and basic parallelization of the algorithm spans roughly the first half of total development time. The second phase starts with the major performance leap described above and then captures mostly minor performance improvement until the end of the project.

While an early analysis of dependencies of student effort-performance pairs to SLOC, programming models and programmer experiences is performed in [Mil16], an in-depth analysis of relations needs to rely on more data sets that further incorporate quantifications of impact factors such as programmers' pre-knowledge. Therefore, quantification and data collection methods are part of my effort estimation methodology and further described in later sections. Finally, the performance life-cycle of the real-world code *ZFS* using OpenACC for GPUs, which was also part of [Mil16], illustrates that the factor Q in Equ. (9.1) can be high for initial cluster environment setup. Corresponding work is discussed in Part D.

10.3. Identification of Impact Factors

One challenge of effort estimation is the dependency of software development effort to numerous impact factors. For example, the COCOMO II model uses a total of 22 factors. Since these COCOMO parameters do not fit well for HPC projects (see Sect. 9.2.1), impact factors on HPC development effort must be newly and systematically examined.

10.3.1. Ranking by Surveys

For the systematic identification of impact factors on effort in HPC, I start from own experience as well as conversations with HPC professionals. The corresponding list of impact factors is then posed to HPC developers through surveys asking participants to rank the given factors and to suggest missing ones, both based on their personal experience. As electronic survey providers, I use *LamaPoll* [LMN17] and *SoSci Survey* [LL17] that both provide data privacy protection according to German standards. An overview of the ranking survey is also provided in a spreadsheet on our webpage [Wie16a]. When ordinal ranking is applied, developers may refer to worst-case or best-case scenarios yielding great variability between impact levels of factors. For example, in an otherwise-unchanged HPC setup where only the compiler technology differs, a user of a novel compiler for parallel programming directives might encounter numerous compiler bugs and sees much impact on effort (worst case), whereas a user with a mature compiler does not see any impact at all (best case). To compensate for this, I create a ranking baseline across all factors that assumes the respective worst-case scenario for the factor as experienced by the developer. Some possible worst-case scenarios are shown in Tab. 10.3.

Table 10.3.: Selected factors impacting development effort.

	Factor	Worst-case example
KP	Pre-knowledge on hardware & parallel prog. model	no pre-knowledge
KA	Pre-knowledge on numerical algorithm used	no pre-knowledge
CW	Code work	starting from scratch instead of using libraries
HW	Architecture/ hardware	programming for very specialized hardware
PC	Parallel prog. model & compiler/ runtime system	low-level programming with intrinsics or experimental compiler with many bugs
TL	Tools	no tools like debuggers available
PF	Performance	code shall succeed certain performance limit
EE	Energy efficiency	code shall be highly energy efficient
AL	Kind of algorithm	algorithm contains many dependencies instead of being embarrassingly parallel
CS	Code size	parallelization of a huge code needed
PM	Portability & maintainability over code's lifetime	code must be maintainable for > 20 years and therefore able to run on different architectures

An initial list of impact factors included 18 items and was handed out to participants of an one-day hackathon at LLNL [Law15]. While results of this survey are illustrated in my SC15 poster [WCMS15], I reduced this list to 11 factors since it was deemed to long by survey takers. I derived a common set of parameters (depicted in Tab. 10.3) through further interviews with various HPC experts. However, these interviews also revealed that the importance of factors varies across HPC developers and that not all possible impact factors can be covered in a survey. Therefore, it is noteworthy that the factor list in Tab. 10.3 is not supposed to be the final list and instead symbolizes the application of my methodology to identify and rank factors. Free form fields in the surveys allow to specify missing factors, and (statistical) analysis can expose factors that depend on each other or do not really contribute to effort and thus can be reduced or combined. Final or intermediate factor lists reveal key drivers on development effort represented by the most highly-ranked factors. Follow-up quantifications of HPC impact factors are supposed to focus first on these key drivers as they will have the highest impact on effort estimates.

10.3.2. Related Work

As elaborated previously, I assume that impact factors on effort for HPC projects differ from those identified for COCOMO II. For COCOMO II, McConnell [McC06] computes the influence of the parameters by dividing the largest numerical value

by the smallest one. Table 9.4 presents the COCOMO factors in order of their significance. Thus, for example, a team with low application experience (APEX) needs $1.51\times$ more effort than a team with high APEX. While this comparison is straightforward for COCOMO's effort multipliers, the influence of the scale factors depends on the code size. For projects of 100 KLOC, all scale factors are in the lower half of significance and move to the upper half for, e.g., 5,000 KLOC (compare [McC06, p. 71]).

Impact factors on software development effort in HPC have not been studied systematically so far to the best of my knowledge. In contrast, HPC development effort is mainly examined as part of comparisons of parallel programming models (as in [ESEG⁺06][CES12] and further discussed in Sect. 10.5.2). Besides the parallel programming model, I experience other factors to also be important such as the programmer's experience and knowledge. However, these are often simplified or ignored in other works (see Sect. 10.4.2).

While my approach is based on ranking factors relative to each other, a more common way in surveys is to ask for importance of factors by using Likert-type items. Exemplary Likert scales are given by the rating scheme of COCOMO's cost drivers going each from *very low* to *extra high*, or by the evaluation process of mainstream consumer acceptance of certain features [MJ77] where importance of an attribute is surveyed from *slightly* to *extremely* important and performance from *fair* to *excellent*. However, the interpretation of results coming from Likert-type scales are controversially discussed in the community, as also a literature review and evaluation by Norman [Nor10] shows. One controversy is between Likert scales mathematically delivering ordinal data and their interpretation as interval data for all practical purposes in social sciences. To stay away from the controversy and since I do not rely on interval information for the factor ranking, I retain my previously-explained ranking procedure. Moreover, the selected ranking method avoids the potential risk that survey takers end up rating all factors as important and thereby diminish the usefulness of survey results.

Finally, after identification of appropriate impact factors, my methodology suggests to concentrate on the key drivers first. This is similar to key driver analysis [Sam01] or importance-performance analysis [MJ77] known from market research.

10.3.3. Statistical Methods

As motivated in the previous section, ranking results from my surveys generate ordinal data across factors for each survey taker. That means that each ranked factor is interpreted as single variable with values between 1 and the total number of factors (here currently 11) from different participants. In statistics, this is called a repeated-measures design with 11 levels.

First, I evaluate whether the impact levels differ significantly across factors to rely further analysis on statically meaningful data. For that, I apply the Friedman test described in [HS16, pp.618] which meets the constraints of ordinal data in a repeated-measures design. For p -values with $p < 0.05$, the null hypothesis is rejected and it can be concluded that the given factors differ significantly.

Second, the ranking order (high impact vs. low impact) is investigated with a multiple pairwise comparison of factors. For that, the one-sided signed Wilcoxon rank test [HS16, pp.542] with inclusion of ties and Holm correction (as posthoc test) is applied [HS16, pp.570]. The one-sided signed Wilcoxon rank test evaluates significant difference between an higher and lower ranked factor. The Holm correction is needed to control the familywise error and to avoid a build-up of type I errors¹ across pairwise tests. Factors ranked very low could be considered to be removed in future ranking studies or replaced by missing factors.

Third, the effect sizes between factors are computed by a series of Wilcoxon tests [Fie13] to give indication on how much factors differ among each other.

While the above statistical methods have shown to be valuable in a proof of concept study, applying further statistical analysis is conceivable. This may include correlation and cluster analysis to combine or eliminate certain factors.

10.3.4. Proof of Concept

For a proof of concept, I have surveyed 20 HPC professionals and 24 students at events marked in Tab. 10.1 by “EF”. The Friedman test shows that the impact level changes significantly across factors ranked in all 44 data sets: $\chi_F^2(10) = 154.57, p < 2.2 \cdot 10^{-16}$. The order of factor importance is derived by a simple rank sum combined with the one-sided Wilcoxon rank test. Figure 10.3 illustrates the corresponding order with factors of high impact on the top and factors of low impact on the bottom. The pairwise p -values are shown in the upper triangle matrix and must be read as follows: row factor is significantly higher ranked in impact as column factor if $p < 0.05$ (marked in grey). Effect sizes are illustrated in a transposed manner in the lower triangle matrix. The analysis of current p -values does not yield a clear winner with respect to highest impact, thus, I use rank sums and effect sizes as indicator until more data sets are collected. The first five factors are: pre-knowledge on hardware/programming model (KP), code work (CW), pre-knowledge on algorithm (KA), programming model/compiler (PC) and performance (PF). They should be emphasized first in follow-up studies. In contrast, current data reveals clear low-impact candidates with medium- to large-size effects: kind of algorithm (AL), code size (CS), energy efficiency (EE)

¹Type I errors describe the incorrect rejection of a true null hypothesis, while failing to reject a false null hypothesis is a type II error [HS16, p. 427].

	p-values											Rank sum
	KP	CW	KA	PC	PF	HW	TL	AL	CS	PM	EE	
KP		1.00E+00	2.91E-01	3.14E-01	1.66E-01	6.90E-02	7.36E-02	3.51E-04	1.71E-04	3.87E-08	1.63E-10	161
CW	6.38E-02		7.61E-01	4.91E-01	4.88E-01	7.36E-02	2.48E-01	2.40E-04	2.26E-06	1.76E-07	1.10E-08	175
KA	2.35E-01	1.68E-01		1.00E+00	1.00E+00	9.62E-01	1.00E+00	1.97E-02	5.32E-04	4.48E-07	7.21E-08	206
PC	2.29E-01	1.96E-01	9.37E-03		1.00E+00	1.00E+00	1.00E+00	4.81E-02	5.32E-04	1.44E-07	5.66E-10	211
PF	2.61E-01	2.02E-01	7.05E-02	9.00E-02		1.00E+00	1.00E+00	2.46E-01	7.39E-03	5.45E-06	1.84E-10	227
HW	2.95E-01	2.91E-01	1.51E-01	9.15E-02	2.63E-02		1.00E+00	3.75E-01	1.42E-02	2.97E-05	5.56E-08	235
TL	2.91E-01	2.43E-01	1.06E-01	1.22E-01	5.18E-02	3.15E-02		4.88E-01	6.82E-03	6.67E-06	5.56E-08	238
AL	4.32E-01	4.40E-01	3.34E-01	3.07E-01	2.45E-01	2.17E-01	2.00E-01		3.14E-01	3.19E-03	4.90E-04	296
CS	4.46E-01	5.15E-01	4.23E-01	4.23E-01	3.61E-01	3.44E-01	3.64E-01	2.29E-01		5.16E-01	3.14E-01	348
PM	5.62E-01	5.47E-01	5.35E-01	5.48E-01	5.02E-01	4.76E-01	4.99E-01	3.83E-01	1.91E-01		1.00E+00	393
EE	6.03E-01	5.74E-01	5.55E-01	5.96E-01	6.02E-01	5.59E-01	5.58E-01	4.25E-01	2.30E-01	5.32E-02		414

Figure 10.3.: Results of one-sided signed Wilcoxon rank test with Holm correction based on 44 data sets. Upper triangle shows p -values (values with $p < 0.05$ in grey). Lower triangle is transposed and shows effect sizes with shade levels $r > 0.1$, 0.3 and 0.5.

and portability/ maintainability (PM). For future investigations, these factors might be candidates to be excluded or at least to be delayed.

Results on impact levels distinguished by HPC professionals and student data can be found in Fig. A.3a and A.3b in Appendix A.3.2. While both groups have AL, CS, EE and PM in the lower third of importance, the impact levels of the remaining seven factors differ across groups. Data from HPC professionals is roughly in line with the overall result in Fig. 10.3 having CW, KA, KP and PC up front. Contrarily, students rate PC and KA as less important factors on effort. Given that most of them have been confronted with HPC as part of a short-term programming assignments with pre-defined (simple) algorithms and a well-tested cluster environment, this result seems reasonable. Future studies will show how much impact factors differ across different groups.

A comparison of overall results of HPC impact factors with similar matching factors in COCOMO II (see significance rating in Tab. 9.4) shows similarities in the relatively low-ranked factors PM and COCOMO II's RUSE. However, similar matching factors like KP and COCOMO's LTEX convey clear differences in HPC and SE cost drivers with KP relatively much higher ranked. This supports the initial assumption that impact factors from SE do not easily translate to HPC.

10.4. Quantification of Impact Factor “Pre-Knowledge”

Given the rankings of KP and KA in Fig. 10.3, pre-knowledge is among the key drivers of development effort. With pre-knowledge, I refer to HPC-related knowledge and experience prior to the begin of development. Since pre-knowledge has an

Table 10.4.: 3-part rating scale for knowledge surveys, similar to [WP05].

A	I am confident that I can adequately answer the question for graded test purposes at this time.
B	I can now answer at least 50 % of the question or know precisely where I can quickly get the information needed and return here in 20 min or less to provide a complete answer for graded test purposes.
C	I am not confident to answer the question sufficiently for graded test purposes at this time.

impact on the developer’s learning curve and capabilities, it also affects development time. Besides being of particular interest in modeling software development effort, investigation of differences in human pre- and post-knowledge is also relevant for organizations in industry and academia to assess and improve training activities, dissemination efforts or tool designs.

10.4.1. Knowledge Surveys

To quantify pre-knowledge as factor in the performance life-cycle, I use so-called *knowledge surveys (KS)* that greatly reduce subjectivity in ratings as problematic with COCOMO’s approach (compare Sect. 9.2.1). “Knowledge surveys provide a means to assess changes in specific content learning and intellectual development” [NK03] by evaluating results in two stages: from pre- and post-KS. Knowledge surveys encapsulate a set of knowledge questions and survey takers are supposed to rate their level of confidence in answering each question. For that, a three-point rating scale is available as given in Tab. 10.4. Corresponding responses can be encoded as interval data since distance information is inherit in the data: To be in-line with other works [NK03][BBH05], I use as numerical value 3 scores for answer A (= 100 % confidence), 2 scores for answer B and 1 score for answer C. Since the focus of my methodology is not on the learning progress, but on insights on pre-knowledge, I only exploit results from pre-KS. The quantification of (pre-)knowledge is then represented by the average of ratings across all answers, eventually distinguished by certain subject fields.

Since survey takers do not really answer the questions in terms of content, but only in terms of their confidence level, they are able to work through the KS questions quickly. Thus, typical KS contain 200 to 300 items. In contrast, I have created knowledge surveys with only 40 questions, i.e, equally amounted as in [FWW14], to reduce overheads in taking the survey, but retaining statistically reliable numbers (also compare [NK06]). Practically, this results in answering times of roughly 30 minutes. Because this is still not a trivial amount of time,

I try to provide an incentive to participate. For students, I created KS with questions that give an impression of possible contents of their oral exams. Thus, taking pre-KS can be seen as exam preparation. To motivate students to also participate at post-KS, that I use for verification of the KS setup only, I provided a book raffle across all students who completed both surveys. Contrarily, HPC professionals cannot be easily motivated to invest the overhead for taking surveys, only if they are personally interested in the research of knowledge effects. In future, I will investigate whether real-time comparison to other votes increases incentives for HPC developers.

According to the design of KS, questions should adhere to Bloom's taxonomy or similar levels of reasoning [NK03]. Bloom [Blo56] defined the six cognitive levels of knowledge, comprehension, application, analysis, synthesis and evaluation. To the best of my knowledge, I provide the first application of Bloom's levels with respect to the HPC domain. Concerning the question distribution to Bloom levels, I stay as close as possible to Nuhfer's and Knipp's suggestions [NK03] resulting in 7.50 % for each knowledge and comprehension, 15 % for each application and analysis and 7.50 % for each synthesis and evaluation. Although knowledge surveys target at confidence ratings, I ask students to also actually answer six of the questions (one from each Bloom level) at the beginning of the survey. This increases the ability of students to correctly judge the complexity of the different types of questions and, thus, improve the accuracy of their ratings. For HPC professionals, I assume that they have gained sufficient self-efficacy capabilities over time so that their confidence ratings are adequate and no additional overhead is needed to really answer questions.

Finally, pre-knowledge results must be set in relationship to the performance life-cycle. For that, corresponding data pairs must be initially collected by the same human subjects consisting of effort-performance logs and knowledge surveys.

10.4.2. Related Work

The impact of pre-knowledge and experiences on effort in HPC has been hardly investigated or otherwise simplified. COCOMO II provides the knowledge factors LTEX and APEX, but only uses years of experience as rating scale (see my discussion on challenges in Sect. 9.2.1). Furthermore, Christadler et al. [CES12] simplified assume that previous experience across developers using different programming models is similar. They do not further quantify experience and just encapsulate it in having "prior HPC knowledge" and "no or very little experience with the language". In [HCS⁺05], the authors also assume "similar" experience levels of student developers doing parallel programming classroom assignments. They base their argumentation on survey results where they asked students in a

three-point rating scale on prior experience: *none*, *classroom* or *professional*. Another simplification of dealing with experience levels are so-called within-subject comparisons, i.e., the same group of subjects participates in more than one (effort) study using the assumption that their knowledge remains constant across studies. This is considered by Kepner [Kep04a] who assumes the same programmer to implement an MPI and OpenMP program, as well as by Hochstein et al. [HCS⁺05] who let students first write MPI code and then an OpenMP version of the same application. However, the within-subject setup does not account for any learning effects, i.e., finding a solution for a problem with one approach will also help solving the problem with another approach (potentially with less effort). This threat of validity is also noted by Hochstein et al. [HCS⁺05] and the reason why I created different reference groups in my student software lab studies, each having a different order of applying three parallel programming models to as serial code base (see Appendix A.1.3). In [HBVG08], Hochstein et al. compare development times of two student groups each using a different parallel programming model for the implementation of sparse-matrix dense-vector multiplications. The authors extend their effort evaluations by asking students about their backgrounds in terms of their major, and experiences in general software development, in parallel programming, in multithreaded programming, in C development, in C++ development and in sparse matrix methods. However, their data does not reveal any statistically significant effects of these factors on development time employing a one-way analysis of variance (ANOVA).

The concept of knowledge surveys dates back to the early 2000s. They are usually applied in classroom environments to compare the success of different technical or pedagogical teaching methods. Moreover, they are used to evaluate the course design revealing potential inconsistencies between subject material and examination contents. They can even help planning goals at a curriculum level by emphasizing agreement in multi-section and multi-instructor courses. Knowledge surveys have been conducted in student classes in biology [BBH05][FWW14][CG12], geology [WP05], chemistry [BV11] and maths [CG12]. However, no prior work covers KS in a programming context. The challenging interpretation of Bloom’s taxonomy for (parallel) programming might be one reason for that. Helpful are few works that present examples of Bloom levels in programming: For instance, Khairuddin and Hashim [KH08] present some small examples based on C for the assessment of software engineering courses, and Gluga et al. [GKL⁺12] provide an online tutorial to train educators and professors in the application of Bloom’s taxonomy in computer science. Nevertheless, Thompson et al. [TLRW⁺08] also show that the assignment of Bloom levels to computer science tasks can be ambiguous, especially depending on the learning contents of the course. Although, I also see this challenge in the application of Bloom’s taxonomy to knowledge surveys, I assume that, due to the high amount of questions, single potentially misinterpreted tasks do not make a huge difference in the overall evaluation. Finally, to

the best of my knowledge, I provide the first translation of Bloom’s taxonomy with emphasis on HPC. To ease the use of using these questions in further knowledge surveys or other courses, I provide an overview and categorization by Bloom publicly available on our webpage [Wie16a].

Despite the application of KS in several fields, their ability to use self-evaluation as an accurate measure for knowledge is discussed [CG12][BV11]. Favazzo et al. [FWW14] show by letting a control group of students rate their confidence and really answering the KS questions that they have a tendency to overconfidence when not really answering questions. Therefore, I established the six-question introduction in student KS where students do both, confidence rating and answering knowledge questions.

10.4.3. Statistical Methods

The KS confidence ratings are interpreted as interval data so that meaningful averages can be easily computed. These numerical values can be used for the regression analysis in the context of the performance life-cycle. Furthermore, their impact on the figure shape of effort-performance pairs can be graphically or analytical be evaluated. For example, the turning point of each effort-performance function can be determined and correlated to the pre-knowledge result.

Furthermore, an early validity check for the design of the knowledge surveys is appropriate. For that, the one-sided signed Wilcoxon rank test can be applied to reveal whether participants show a learning effect between pre-KS and post-KS.

10.4.4. Proof of Concept

I have created several different knowledge surveys for GPU hackathons and student software labs that I also provide online [Wie16a]. Questions of the KS for hackathons cover the application under investigation, general parallel programming and computer architecture topics, as well as CUDA and OpenACC knowledge with respect to GPU usage. Software lab KS ask students about the CG Solver, general HPC topics like Amdahl’s Law, (ccNUMA) CPU and GPU specialties, and about programming with OpenMP, OpenACC and CUDA. The corresponding KS are provided electronically: For hackathon participants, I used SoSciSurvey [LL17] as survey provider, and for students, I used the survey capabilities of the RWTH’s e-learning platform to restrict participants to the corresponding class. I set the starting time of the fill-in period of the pre-KS to approximately one to two weeks before the event starts and I close participation time at the end of the event’s first day. With that, I allow flexibility in time when to take the survey, but can

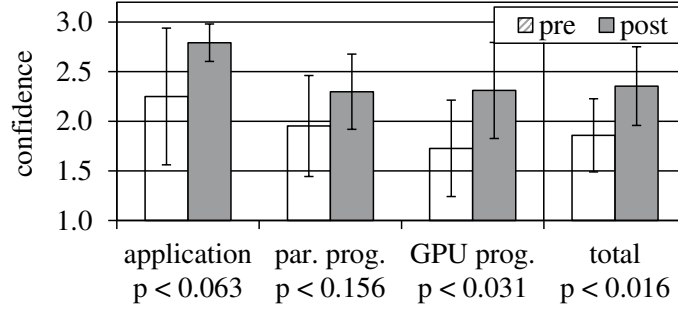


Figure 10.4.: Means of pre-KS and post-KS results per knowledge question group based on 6 data sets. Error bars indicate standard deviation.

also validate that answers correspond to prior knowledge. The additional first day during the event is needed, to remind and motivate participants to take the KS.

As evident in Tab. 10.1, I got four matching pre- and post-KS from hackathons and two matchings surveys from the student software lab in 2016. With these results, I test the significance between pre- and post-KS to show the validity of my surveys. The overall rank sums show significant differences at a 5% level with $p < 0.016$. Because of the small sample size, I add descriptive information and plot the means across different questions groups. Fig. 10.4 illustrates an average increase of confidence per knowledge group. Results distinguished by HPC professionals and students can be found in Fig. A.4a and A.4b, respectively, in Appendix A.3.3. They indicate that the learning effect of students is much higher than of professionals during the given events.

Furthermore, pre-KS ratings from the software lab indicate that a simplified assumption that all students have similar pre-knowledge cannot be hold. Although our students take part in an introductory workshop on parallel programming before taking the pre-KS, variability in students’ knowledge does still exist. With a two-sided signed Wilcoxon rank test across students and their pre-KS confidence ratings, it can be shown that these differ significantly for 12 out of 21 possible stu-

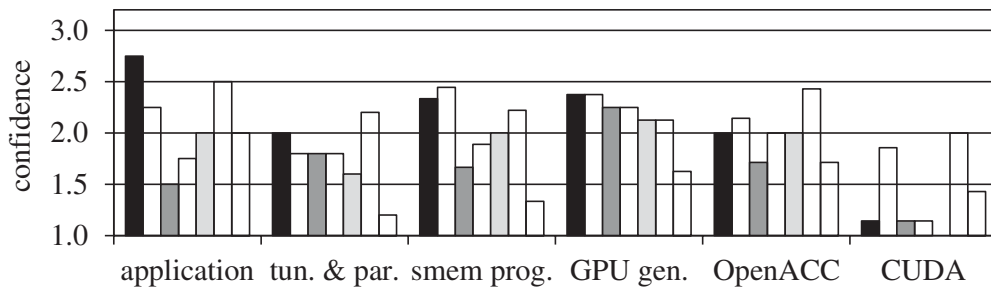


Figure 10.5.: Variability of pre-KS means per knowledge question group across 7 student data sets.

dent combinations (at a 5 % level). Descriptively, this is evident in Fig. 10.5 where each hatched bar represents ratings of one participant. Given values are average ratings for the specified knowledge question groups: Here, *tuning & parallelization* and *shared memory programming* equals *parallel programming*, and *GPU general*, *OpenACC* and *CUDA* are combined to *GPU programming* in Fig. 10.4.

10.5. Quantification of Impact Factor “Parallel Programming Model & Algorithm”

In addition to pre-knowledge, the parallel programming model plays an important role in affecting effort needed for employing HPC activities to scientific application. It has been of great interest in numerous works and its importance is also visible in the ranking order in Fig. 10.3. Although not apparent in this small data set, empirical experience leads to the assumption that the impact of the parallel programming model is strongly related to the application to be parallelized. For example, in [WTBM14], I show different expressiveness of OpenACC and OpenMP depending on the algorithmic pattern on hand. Consequently, parallel programming models can be more or less suitable to implement certain (parallel) patterns contained in the application’s algorithm.

This section differs from the rest of the work, since I sketch ideas rather than having case studies with applicability results ready yet.

10.5.1. Pattern-Based Approach

The core idea for quantifying the impact of the parallel programming model in combination with the application’s type relies on the decomposition of the application’s algorithm into patterns and a follow-up suitability analysis based on these patterns. I further envision to maintain a *suitability table* that contains general application-overarching information indicating good matches of listed programming models to patterns. Mapping the extracted patterns from the target application to the information of the suitability table reveals input data for the effort estimation equation. However, this approach implies two major challenges: dividing an algorithm into meaningful patterns and setting up the suitability table.

Today, patterns are extracted from the given target application by manual inspection. Patterns must be chosen in a way that they appropriately classify the algorithm. However, typical algorithms do not embrace a single pattern or consists of several distinct patterns. Therefore, a pattern-based algorithm characterization must also be able to deal with overlapping or nested patterns. Although some popular patterns exist in HPC, such as the Berkeley’s dwarfs (or motifs) [ABC⁺06],

skeleton patterns [Col91][MRR12] or design patterns [MSM04], my experiences show that none of them suit the purpose of completely classifying an algorithm. For example, based on my previous work [WTBM14] and the supervised Master thesis of Dammer [Dam14], I conclude that low-level abstracted skeleton patterns are too fine-grained to categorize algorithms and workloads in a meaningful way. Instead, Berkeley’s motifs are too coarse-grained to deal with algorithmic specialties that may strongly affect development time. Since the design patterns promised to have an abstraction in the middle of the former two approaches, I created a questionnaire that asks HPC developers to decompose their algorithms into these patterns, denote possible overlap of patterns and estimate required effort per pattern. However, a first test run accomplished by an HPC developer in the field of molecular dynamics [Haq15] revealed that this pattern abstraction level is not suitable for direct application either. Hence, future investigation will focus on a meaningful pattern abstraction and decomposition, at best in an automated fashion.

Taking the assumption that a pattern-based classification of algorithms is possible, the suitability table must be set up. The idea is to manually compile this table for each parallel programming model and pattern in an HPC-wide one-time community effort. That means if a new parallel programming model emerges, HPC developers extract the corresponding pattern mapping information once to feed it into the table. One part of this mapping information contains the estimated development effort whose specification is based on effort data tracked per pattern. For technical support of the tracking effort, I have started to investigate so-called *pattern instrumentation* capabilities. These can be used by the developer to annotate certain code regions in the application that represent the patterns needed for classification of the algorithm. Development effort spent for parallelizing or tuning the code can then be automatically broken-down per pattern, so that each annotated pattern relates to certain portions of the tracked time. However, the implementation and validation of this concept is still work in progress.

10.5.2. Related Work

With the HPCS program and further activities towards a hardware-software co-design [BBD⁺13], numerous parallel programming models have emerged with different abstraction layers. Therefore, it is not surprising that an extensive list of publications covers the comparison of parallel programming models. As seen in Sect. 9.1, numerous works use basic software complexity metrics to reason about one approach or the other, usually in combination with some performance measurements. However, here, I focus on related work that provides a view on parallel programming models with respect to development time. Ebcioglu et al. [ESEG⁺06] compare student efforts for creating a first parallel version of a Smith-Waterman

local sequence matching algorithm for MPI, UPC and X10. In my previous work [WaMM13], I present efforts for a single programmer for two real-world applications across OpenMP, OpenACC, OpenCL and Intel’s Language Extensions for Offload and put them in a TCO and productivity context. More programming models are investigated in the PRACE project by Christadler et al. [CES12]. They compare performance in relationship to development times by different programmers for three kernels using Chapel, MPI+OpenMP, CAF, CAPS, X10, CellSs, UPC, RapidMind, CUDA, OpenCL, CUDA+MPI. However, statistical evaluation of significant differences between development times with various parallel programming models is rarely covered. Hochstein et al. [HBVG08] compute a 95 % confidence interval for the differences in development time means of two student groups implementing a sparse matrix vector multiplication with MPI and XMTC. They further show that the parallel programming model exhibits a statistically significant effect on development time by using ANOVA. In another work, Hochstein et al. [HCS⁺05] compare OpenMP and MPI in a within-subject student study using a paired t-test. Since the assumption of the t-test having a normally-distributed population was not verified, I use a one-sided signed Wilcoxon rank sum test to compare student efforts for implementing a CG solver with OpenMP, OpenACC and CUDA [WCMS15], similar to Prechelt [Pre00] who applied it to compare C, C++, Java, Perl, Python, Texp, Tcl.

Taking the pattern perspective, numerous terms have been established for the classification of algorithms with potential differences in focus: parallel patterns, design patterns, algorithmic skeletons, parallel building blocks, idioms, templates or workload characterization. Here, I see *parallel patterns* “as recurring combination of task distribution and data access” [MRR12, p.79] rather than as “a good solution to a recurring problem” as Mattson et al. [MSM04] define *design patterns*. Most works in the pattern domain extend the idea of Mattson’s design patterns by introducing frameworks or parallel pattern languages. However, they usually only provide a limited number of pre-defined patterns (extensible by the user), so that these cannot be used for a classification of applications. González-Vélez and Leyton [GVL10] give an overview of different algorithmic skeleton frameworks. The other group of works focus on a characterization of applications using performance metrics. For example, Manakul et al. [MSSA12] assign NAS and Rodinia Benchmarks to different dwarfs by their stalling behavior and Treibig et al. [THW13] define performance patterns such as load imbalance, bandwidth saturation, or bad instruction mix. Nevertheless, three popular pattern approaches exist that could be used for application categorization. First, McCool et al. [MRR12] define structured serial and parallel patterns with very low abstraction level, e.g., map, scan, fork, stencil, gather. In Wienke et al. [WTBM14], I used these patterns to compare the expressiveness of OpenMP and OpenACC for accelerator programming. Furthermore, Dammer [Dam14] applies these for a comprehensive comparison of OpenMP, OpenACC, CUDA and MPI for Intel

Xeon Phi coprocessors and NVIDIA GPUs. Second, the concept of algorithmic skeletons was introduced by Cole [Col91] in 1991 and revised and extended by Mattson et al. [MSM04] who define a parallel pattern language and its design patterns. Ongoing work [KM09][KMMS10] attempts to combine this approach with the following one. Third, Berkeley’s motifs [ABC⁺06] provide a high level categorization by 7 dwarfs (later extended to 13 dwarfs) such as dense linear algebra, structured grids or n-body methods. They are used for classifying algorithms in the PRACE context [SBH08]. Finally, van Amesfoort et al. [VAVS10] state goals similar to mine in terms of application characterization, however, they focus on performance and provide only limited suggestions, instead of a full categorization.

10.5.3. Proof of Concept

Although, I cannot present data that directly supports my pattern-based concept of quantifying the impact of the parallel programming model and application algorithm, I can show that the choice of parallel programming model affects development time in HPC.

While I presented differences in programming models based on student software lab data from 2014 in my SC15 poster [WCMS15], I now extend this evaluation to all available data from years 2013 to 2016 [WCM16]. A total of 25 student teams submitted effort logs and runtime data for all three parallel programming models (OpenMP running on CPUs, OpenACC and CUDA running on GPUs) achieving a numerical correct solution (see Sect. A.1.3). These can be split into years 2013 to 2014, where 14 student teams started with a sparse matrix in CRS format and a roughly predefined implementation order: OpenMP, OpenACC, CUDA, and into years 2015 to 2016, where 11 student teams began with a sparse matrix in ELLPACK-R format in completely permuted order of implementation. This re-ordering in different control groups shall account for potential learning effects when implementing the same parallel algorithm with more than one parallel programming model. Since I cannot assume normal distribution across student teams, I use the one-sided signed Wilcoxon rank sum method to test for significant differences between programming models. Figure 10.6 shows results expressed as p -values,

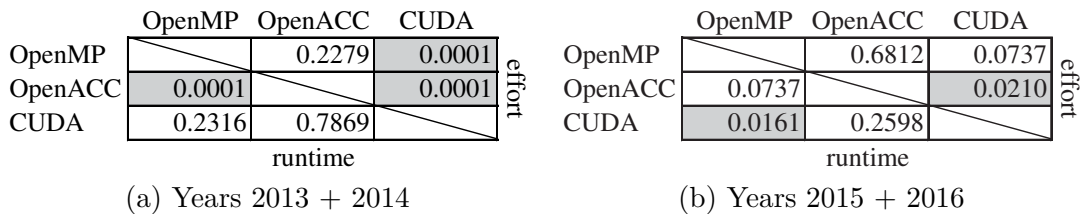


Figure 10.6.: p -values of one-sided Wilcoxon rank sum test with respect to students’ development effort (upper triangle) and runtime (lower triangle).

so that the row item is significantly lower than the column item (on a 5 % significance level). Thus, student lab data from 2013 and 2014 (Fig. 10.6a) indicates that OpenMP and OpenACC both require less effort than CUDA. Furthermore, OpenACC runtimes on GPUs are significantly lower than OpenMP runtimes on CPUs. Due to the changed base matrix format and corresponding impacts on effort for optimization, I do not want to directly compare results from the early years to the later years. However, Fig. 10.6b shows that, despite permuted implementation order, two parallel programming models still have a significant difference in development effort, i.e., OpenACC and CUDA. Since the corresponding implementations are based on the same CG algorithm and the same (GPU) hardware architecture, this difference can be contributed to the differences in programming model themselves.

Hence, although these case studies do not provide any quantification, they still show that the programming model has some impact on development effort - even if the algorithm and hardware architecture remain the same.

11. Data Set Collection

Recapping the (statistical) quantifying concept of my performance life-cycle (compare Sect. 10.2), it relies on the availability of suitable data sets, particularly in content and amount. This is also evident in the numerous proof-of-concept studies described earlier. Data sets need to comprise information on the amount of effort spent for HPC activities, the kind of HPC activity carried out, application's performance, and performance milestones. However, data set collection also faces several challenges that are further discussed in this section.

Similar to the previous section, work presented here is mainly based on Wienke et al. [WMSM16]. Furthermore, the *EffortLog* tool [MW16] is under continuous development by Miller, initiated as part of his student worker activities under my supervision.

11.1. Related Work

Techniques for data collection can be mainly divided into manual measurements, i.e., using developer diaries, self-reports or recordings of observers shadowing the developers, and automatic measurements using instrumented development environments. While Faulk et al. [FGJ⁺04b] give a brief overview of these techniques with some of their strengths and weaknesses, my focus is on discussing challenges during their application.

11.1.1. Manual Diaries

When collecting data by developer diaries, developers are asked to log their activities (usually) in a paper-pencil fashion. The details of inquired information depends on the study (duration) and the developer who decides on the level of reported details and the number of entries. The concept of diaries is likely to stem back to the *Personal Software Process* [Hum95] from SE that describes manual data collection for time, size and quality measures. In HPC, the technique of manual diaries is also commonly used for tracking effort data. Insights into a diary template are provided by Erbacci et al. [ECSC09] in the PRACE context. They asked developers to denote development time, achieved performance, number of

cores, data sets and additional comments on the HPC activity and problems faced. Hochstein et al. [HBZ⁺05] also reveal their structure of diaries: They started with a web-based interface that asked for development time, and gave a selection of HPC activities spent since the last entry. They updated this approach into a paper-based diary because they suspected that the web-based form was one reason for underreporting. Furthermore, they switched from asking for development time in hours to start and stop times with the aim of increased precision. Finally, they added “breaks” to the selection of HPC activities. Based on these works, Cramer and I [WCM16] established our developer diaries distributed to students in software labs. We asked for development time, number of modified LOC, achieved performance (in seconds), and porting and tuning steps and whether students encountered any problems. Due to the students’ unclarity on what to report, I added a corresponding description to the diary template.

The main challenges that arise from keeping manual developer diaries include overhead time, inconsistent data and inaccuracy. My experience from working with students and professional developers shows that developers need a high level of commitment and diligence to provide complete, consistent and correct information, even under time pressure. Similar observations are depicted by Perry et al. [PSV95] who investigate the development process of professional developers in building a real time switching system, i.e., traditional software engineering without focus on HPC. They examine the accuracy of daily time diaries using a direct observation study with five developers on five days and find a fidelity, i.e., the agreement rate between self-reported and observed activities, of 58 % to 95 %. The source of disagreement is mainly attributed to unrecorded unexpected interruptions. Furthermore, Perry et al. find that a higher number of diary entries per day leads to a higher fidelity. Their overall comparison of working times per day delivers an average overestimation in self-reports by 2.80 %. For development in HPC, Hochstein et al. [HBZ⁺05] also compare self-reported and observed data, here during writing and parallelization of artificial kernels by one student for two hours and one professional for nine hours. They see an overestimation of 13 % for the perfectly-conditioned student experiment and a small underestimation of 1 % for the professional developer. Perry et al. and Faulk et al. additionally state the challenge that self-reports may be distorted sheerly due to an (unconsciously) modified development process when keeping diaries. One reason for that could also be social or political pressures.

11.1.2. Automatic Tracking

To get comprehensive, consistent and more precise information on development effort with less overhead and intrusion, several tools have been introduced that follow the technique of automatic data collection usually by capturing certain activities in

an instrumented environment. For example, for SE activities, GRUMPS [TKD⁺03] and Ginger2 [TMN⁺99] track very low-level user information such as window activities, mouse and keyboard events, or even data from eyetracking and skin resistance. In the HPC field, the most-used tracking environment is Hackystat [JHA⁺03]. It proposes sensors that represent plug-ins for attachment to development tools such as editors or debuggers. These sensors log activity data transparently and automatically every 30 seconds and centralize it at a web server. Based on this data, coding and debugging time gets estimated. Similarly to Hackystat, PROM [SJSV03] also provides plug-ins with additional language support and data views for managers. Since most collection of effort data has previously been initiated by DARPA's HPCS program, most tool support adaptations for HPC purposes were also done in this context [DGH⁺08][BZ09]. HPCS studies collected Hackystat data for the investigation of the HPC development workflow [ZH09][HBZ⁺05]. Extended with a Unix shell logger called UMDinst, Hackystat was integrated into the Experiment Manager [HNS⁺08] that collects data on effort, defects, workflow and source files. The Experiment Manager is based on a web interface from which developers manually select pre-defined (HPC) activities and where each activity switch triggers a logging stamp. However, at time of writing, this tool is not available anymore.

Challenging factors in tool-based automatic data collection comprise the type and interpretation of measured data, the setup of instrumented environments, and further political and social issues. While automatic data collection provides objective measures, it focuses on easily-measurable information instead of actually-required information. Therefore, it captures mainly coding activities and neglects effort spent for thinking, training or in-person interruptions. The Experiment Manager includes these activities in its web interface, however, developers need to remember to manually denote any activity switches. That is a risk the authors never elaborated on. Hochstein et al. [HBZ⁺05] further show the difficulty to correctly interpret low-level fine-coarse (Hackystat) data and that current approaches still yield differences in tool-derived effort data compared to directly-observed data. Another challenging task is the setup of instrumented (Hackystat) tools. From my experience, getting the developer's coding and execution environment shaped for automatic data collections is laborious. Moreover, not every coding and development tool is amenable for instrumentation by Hackystat sensors [JHA⁺03]. Consequently, developers might be constrained to a limited tool usage resulting in low programming productivity, or high setup overheads might prevent data collection completely. Finally, non-technical challenges with respect to political and social situations can occur due to the transparency of automatic data collection, its invisibility to the developer and data storage and analysis on (U.S.) web servers. Developers might feel uncomfortable and under surveillance [Joh13] with the consequence of changing their development process, suffering under a negative working atmosphere or raising objection against effort log participation. In addi-

tion, industrial setups might push for more control on their data, and German laws on data privacy protection ask for a decentralized and secure storage of sensible data.

11.2. EffortLog — An Electronic Developer Diary

To tackle the challenges of manual diaries and automatic low-level data collection, Miller and I have developed the tool *EffortLog* [MW16] that serves as electronic development diary. It regularly asks for developer input and supports the collection of information on development time, HPC activities, performance-related development milestones, and performance data and, thus, is one basic element of my methodology for development effort estimation in HPC. A direct comparison of characteristics of manual diaries, automatic data collection and *EffortLog* can be found in Appendix A.4.1.

11.2.1. Tool Description

EffortLog is written in C++ and designed for easy setup with cross-platform, open-source character. Developers just download the sources or releases from Github and start an instance of the tool on their local computer or cluster environment. To account for privacy protection and to guarantee full data control by the developer, it stores all collected data locally and can further be configured to feature encryption and full anonymity of its users. In addition, *EffortLog* is based on standard human-readable JSON input and output files to prevent transparency and, thus, increase social acceptance. On the downside, data analysts rely on the developers' consciousness to convey completed effort log files.

Using *EffortLog* for software projects (e.g., given by “myproject” in Fig. A.5a), the developer initially specifies a sampling interval that might be later adapted. At the end of the interval time, the tool moves to foreground and requests user input (see Fig. A.5c). If continuous development is preferred by the developer at this stage, they can decide to skip the input and respond later. Either case, *EffortLog* tracks the correct interval time since the last input. Following this interval-based approach enables the tradeoff between frequent interruptions and entries with high resolution. First experiences show that log files created with intervals of 30 min to 60 min deliver reasonable accuracy with small overhead. Furthermore, the tool's character of actively asking for input keeps the developer reminded to track their HPC activities so that they do not encounter the risk of forgetting entries as it was possible with the Experiment Manager. To foster flexibility in effort logs and inclusion of thinking or reading time not spent close to the computer, the tool provides capabilities to append logs and to evoke logging any time.

For each log event triggered by the tool, development time is tracked automatically. Moreover, input on the conducted HPC activity is requested that can be selected from some pre-defined categories similar to Hochstein’s et al. [HBZ⁺05] (see Fig. A.5c). Additional activity details can be denoted in free form, may include information on the compiler (flags), the parallel programming model, or the hardware, and, thus, simplify the interpretation of collected data. To extend effort logs with performance data as part of my PLC evaluation, the log event also asks for comments on possible performance-related milestones such as reaching a(nother) tuned parallel version of the software (compare Sect. 10.2.1 for possible milestones). Performance data includes details on the execution time, number of threads or processes, hardware and compiler information, and remarks on the programming model and data set (see Fig. A.5b).

Finally, to improve acceptance of (voluntarily) tracking development efforts, *EffortLog* also provides benefits for the developer: A structured view of logged data including date, HPC activity, duration of development time and performance result (see Fig. A.5d) may help developers to easily grasp activities that improved or decreased performance and to improve their sense of time for these activities.

11.2.2. Proof of Concept

The *EffortLog* tool has been used in the context of a five-day hackathon [TUD16], during one student software lab, and during the tuning process of *ZFS* for GPUs (compare Part D). To evaluate *EffortLog*’s usefulness, I compare electronically logged entries from *EffortLog* to collected data from manual developer diaries for the hackathon event and the software labs in 2015 and 2016.

For the hackathon, one professional developer voluntarily used *EffortLog* gaining a total of roughly 14 hours logged distributed across 14 entries (excluding 3 entries with breaks). From that, 2.30 hours (=4 entries) were spent for thinking. The resolution of the logs ranges from 20 to 111 minutes. During the same event, a manual diary was kept by another hackathon team of professional developers on a daily retro perspective who combined efforts of all four team members into one log. They reported a total of 75 hours in 21 entries with a resolution range of 0.50 to 6 hours (except one entry with 5 minutes), generally 57% of their entries have a resolution of more than 2 hours. For the HPC software lab case study, student teams participating in 2015 used manual developer diaries while student teams in 2016 used *EffortLog*. Each student logged an average of 45.40 hours for all three programming models together in 2015, and an average of 51.72 hours in 2016. The resolution of logging events ranges from 0.03 to 8 hours in 2015 and from 0.07 to 3.57 hours in 2016.

While having noted the risk that developer reporting styles can differ significantly [PSV95], I investigate the number of entries per developer and logged hour

Table 11.1.: Comparison of tracked effort data using *EffortLog* and manual diaries for one hackathon event and two years of software labs. For the software labs, numbers are averages across teams and across the three used programming models.

	hackathon		software labs	
	EffortLog	man. diary	EffortLog	man. diary
#teams	1	1	6	5
logged effort/ developer	~14 h	~19 h	~17 h	~15 h
#entries/ logged hour per developer	~0.98	~0.28	~1.79	~0.22
#milestones/ team	-	-	~2.72	~6.47

as indication for accuracy. Results from both the hackathon event and the software labs in Tab. 11.1 illustrate the tendency to create more entries per hour using *EffortLog* as with manual diaries. For the software labs, the numbers are averaged across collected data with all three parallel programming models. For 2015 data (manual diary), the metric entries-per-hour ranges from 0.201 for OpenMP, over 0.196 for OpenACC, to 0.260 for CUDA. For 2016 data (*EffortLog*), it ranges from 2.10 for OpenMP, 1.56 for OpenACC and 1.71 for CUDA. Furthermore, the maximum logged interval when using *EffortLog* is roughly halve of the maximum interval from manual diaries, for both hackathon and software lab logs. This emphasizes a reduced chance of forgetting short-lasting HPC activities and, hence, increases the chance for accurate results.

From the interval definitions and the actual logged interval times in the *EffortLog* files, it can also be concluded that postponing effort log entries is a good option if continuous work is necessary. Furthermore, the activity categories in the log files also reveal that time for thinking or testing was included and, thus, the tool is capable to capture non-coding activities (in contrast to Hackystat). These categories further improve the interpretation of denoted free-form comments and reduce the developer’s overhead for activity logging. However, when looking on the software lab data, the average logged number of performance-related milestones per student team decreased using *EffortLog* over manual diaries (Tab. 11.1). Due to evaluating data from earlier years, I could not get details for according reasons. However, for more precise effort-performance pairs, I will improve on promoting the usage of milestones. Furthermore, to reduce the overhead in reporting milestones in our tool, Miller and I have technically extended the tool to take over previous definitions of hardware and programming model in any milestone declaration so that obtained performance might be the only number to be adapted.

In future, feedback from *EffortLog* users will be continuously considered and integrated into the tool if applicable. Previous feedback comprises some issues complicating using the tool: forgetting to start it, overhead in logging, and work

not done close to the computer. These issues were tackled by including technical details to minimize fill-out time, and the tool's append capability for delayed collection of effort data. Contrarily, feedback also said that logged data helps keeping track of invested time and applied tuning knobs. Future extensions of the tool may contain additional categories such as setup time or planning, and coverage of multiple team members in one log.

11.3. Challenges

“Collecting data has and continues to be one of the biggest challenges in the software estimation field” is postulated by Boehm et al. [CBS99]. Especially, data collection that involves human subjects inherently face certain challenges and threats to validity as shown in the following. I categorize them into trust in results, and gathering (a sufficient amount of) data. Some additional challenges are summarized by Sadowski and Shewmaker [SS10].

11.3.1. Trust in Data

Reported vs. Observed Data Effort-performance investigations are based on reported efforts from developers using (electronic) diaries. However, discussions in related work (Sect. 11.1) illustrate that developers keeping self-reports in form of manual diaries tend to over (or under) report their efforts compared to recordings from direct observers. Although, tools such as *EffortLog* potentially reduce the error, it will still affect results. Therefore, uncertainty and sensitivity analysis is important when dealing with reported efforts. Correspondingly, I include this into my productivity evaluation (compare Sect. 4.6) and further disturb estimated efforts with typical over and under reporting errors taken from literature.

Side Effects Development effort depends on numerous impact factors. Consequently, human-subject studies often try to eliminate single effects by keeping one parameter fixed. For example, to work around the impact of pre-knowledge, several studies conduct within-subject comparisons. However, possible subjects' learning effects during the study are often ignored. Hence, studies must be well prepared to avoid side effects, e.g., within-subject studies need to explicitly evaluate knowledge factors or appropriately modify the order of experiments. Because of this, I investigate pre-knowledge with knowledge surveys and permuted the order of experiments in student software lab studies.

Students vs. Professionals Most effort studies are conducted with students because these studies are inexpensive and can take place in a controlled environment compared to professional HPC development with large codes, bigger time spans and potential time pressures. However, the relevance of student results have also been debated. As tradeoff, I first assess concepts covered in my methodology with students in a controlled setup and make changes with respect to success and feedback. Then, concepts are tested with professionals in a rather controlled environment, e.g., during hackathons, before applied to regular HPC development.

Bias and Subjectivity The interpretation of collected data, especially of HPC activities, might be distorted by subjectivity in the data. On the one hand, different developers might assign an activity to different categories when ambiguous. On the other hand, analysts doing follow-up data studies might draw different conclusions. Furthermore, effort data might be subject to bias. Bias can arise, e.g., by the type of developers, the used parallel programming model or hardware architecture, or institutional organizations (in a particular country). Especially, the country bias has not been investigated so far since most other effort studies are conducted within the USA. To reduce organizational and country bias, I aim at a world-wide community effort for data collection (see below). Nevertheless, my currently gathered data does most likely contain bias with respect to the used programming models and others. To initially prove concepts, this is still a valid approach, but must be improved for future studies.

11.3.2. Data Gathering

Setup Overhead Collecting effort data from human subjects includes overhead for study setup. The obvious part is preparation of contents. However, in addition, external legal and social conditions must be considered. For example, in Germany, data privacy protection must be correctly implemented, and in the U.S. human-subject research must be formally-applied for at the organization's Institutional Review Boards (IRB).

Correctness Gathered data sets often suffer from incorrect solutions when results are computed by parallel, tuned or ported code. While this issue also reduces the trust in data when not being able to verify the numerical results, it massively decreases the amount of data sets that can be used for further analysis since performance of incorrect results and respective effort spent are not reliable. Because of this, I verify numerical results with the submitted codes from student software labs and exclude student team data with incorrect solutions.

Motivation of Participants The overview in Tab. 10.1 illustrates that from potential participants of an event only few contributed to effort data collection. Student data sets are most reliable when they are intrinsically motivated to participate by getting grades [WCM16]. Motivation by prizes in programming competitions can lead to higher participation, however, results may greatly suffer from students' tradeoff of thoroughness and careful coding vs. development speed. Motivation of professionals is even more challenging since keeping logs still means overhead for them. I experience that my personal communication and passionate involvement increases their own personal interest and, hence, their willingness to participate. *EffortLog* tries (technically) to reduce overhead as much as possible, and additionally gives the developer the benefit of easily observing applied tuning knobs and their impact on performance. Finally, besides motivating participants, it is also challenging to collect their created log files. Since *EffortLog* explicitly avoids a centralized web server approach, participants need to hand in their data sets manually while analysts simultaneously maintain their data protection.

Community Approach Although I continue to collect data sets, any (statistical) analysis should rely on a great amount of (various) data sets for meaningful results. Therefore, as part of my methodology, I have reached out to interested people and asked for participation. I envision a community approach (similar to the communities of Software Carpentry [Wil16] or Women in HPC [Col16]) and public website to bring together interested developers and managers and combine gathered data sets. To help jump start this, I provide material and tools on our webpage [Wie16a] and share lessons learned with like-minded persons.

Part D.

Making the Business Case

HPC matters! [...] It's finding
signals in the noise.

SC14 HPC MATTERS VIDEO

12. Aeroacoustics Simulation Application — ZFS

Focusing on applicability to real-world HPC setups, I evaluate the application of models and methodologies introduced in Parts A, B and C in a case study covering an aeroacoustics simulation application exploiting hardware at the RWTH's HPC center. I employ previously-developed concepts and methods with respect to software development effort, TCO and productivity. While corresponding productivity indices result from empirically-gathered data, I also give an outlook on estimations to next generation hardware at RWTH Aachen University.

The aeroacoustics simulation application *ZFS* (*Zonal Flow Solver*) pertains to the class of large-scaling applications: It scales up to $\sim 460,000$ cores on a Blue-Gene/Q system, and is part of the High-Q club [SLMS15]. Implemented with C++, the code follows an object-oriented design with more than 260,000 LOC.

OpenACC parallelization and tuning of the Discontinuous Galerkin (DG) solver of *ZFS* for GPUs have been undertaken in the context of a Bachelor thesis [Nic15] and Master thesis [Mil16] with my supervision. Parts of these works have been further published in [NMW⁺16] and [MWSL⁺17]. In the following, I give an overview of the application and its porting and tuning activities.

12.1. Description

The multi-physics simulation framework *ZFS* is developed by the Institute of Aerodynamics of RWTH Aachen University [LSG⁺14][HMS08][SGMS16] and solves the acoustic perturbation equations to predict the acoustic pressure field of flow-induced noise [SLYB⁺17]. Figure 12.1 shows a 2D snapshot of the acoustic pressure field of a round jet simulated by *ZFS* that illustrates the different length scales of sound waves [SL17]. To solve the acoustic perturbation equations, a Discontinuous Galerkin method is applied for discretization, and a Runge-Kutta scheme is used to compute the state at the next time step [SL17]. The DG solver of *ZFS* is targeted for porting and tuning with OpenACC and requires approximately 6,500 code lines to be touched. Its main loop covers roughly 99% of the total (serial) execution time for typical test cases and consists of 8 major kernels that further

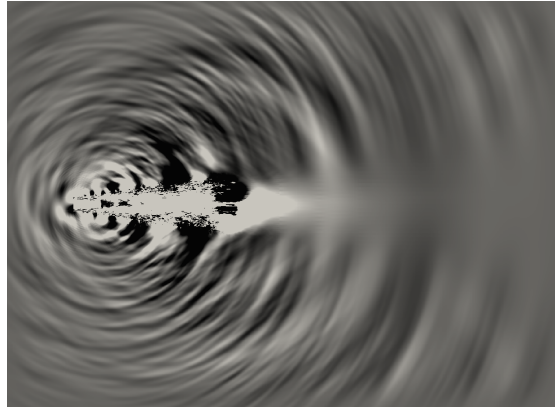


Figure 12.1.: Acoustic pressure field of a jet, taken from [SL17].

comprise a deeply-nested call hierarchy. The most compute-intensive methods include loops over elements or surfaces, and represent the extrapolation of the solution, the flux calculation and integral evaluations.

The simulation test case considered here represents a realistic but strongly scaled-down 3D data set with 32,768 grid cells: A Gaussian-shaped pressure pulse is generated inside a cube and reflected by a solid wall located in z -direction. The simulation is performed for 115 time steps.

12.2. HPC Activities

Based on the CPU-parallel (MPI+OpenMP) version, Nicolini [Nic15] ported the DG solver with OpenACC to GPUs. He parallelized the main methods of the solver by annotating loops over elements and surfaces. For that, typical C++ data access patterns with pointers to underlying arrays had to be removed by working directly on underlying flat arrays, and workarounds for implicit calls to copy constructors that contained non-portable code fragments have been put in place. More details on required code transformations can be found in our work [NMW⁺16].

Taking Nicolini’s first parallel version as foundation, Miller [Mil16] applied numerous GPU tuning steps to the DG solver with focus on the optimization of memory transactions and data access patterns. This includes usage of pinned memory, minimization of data transfers, array privatizations and improved locality for data inside nested loops by exploiting GPU caches. He further applied OpenACC’s streaming concept to overlap various kernel executions, as well as computations and data transfers. Lastly, he added support for multiple GPUs with MPI. In the following, I focus on the first parallel and tuned parallel version, and leave the multi-GPU scenario as future work due to ongoing MPI tuning activities. More information on these HPC activities are covered in our publication [MWSL⁺17].

13. Development Effort

Starting to quantify components of my TCO and productivity model, the effort spent on HPC activities must first be estimated. In this case, HPC activities include the porting of the code to GPUs and its further tuning using OpenACC. Since a complete effort estimation model for HPC purposes is not yet in place, I rather use this case study to illustrate how to apply my methodology to collect data sets. Thus, for making the business case, I rely on actually spent efforts tracked during development.

13.1. Knowledge Survey

Spent efforts on the OpenACC versions of *ZFS* are split across two students that were reasonably-well familiar with OpenACC and GPUs. I investigate and quantify the pre-knowledge of the second developer by creating and distributing pre- and post-knowledge surveys to him. This knowledge survey covers topics in terms of application knowledge, general parallelization and tuning concepts, shared- and distributed-memory programming, as well as GPU tuning techniques and OpenACC specifics. Similar to my other studies (see Sect. 10.4.1), it comprises 41 questions and adheres to the various Bloom levels. I provide most of these survey questions online [Wie16b].

Results from the pre-KS show that the developer is well-versed in all KS areas with an average overall value of 2.73 (compare Fig. 13.1). In comparison, pre-KS results from 6 students attending the RWTH software lab in 2016 delivered an average value of 1.94 within the range of [1.48; 2.6]. Given the high pre-KS result, the one-sided signed Wilcoxon rank test does not show any statistically significant differences between pre- and post-KS values of the developer.

13.2. Performance Life-Cycle

As a next step, the effort-performance relationship is examined. For that, I compare entries from manual developer diaries to ones tracked with *EffortLog* (see

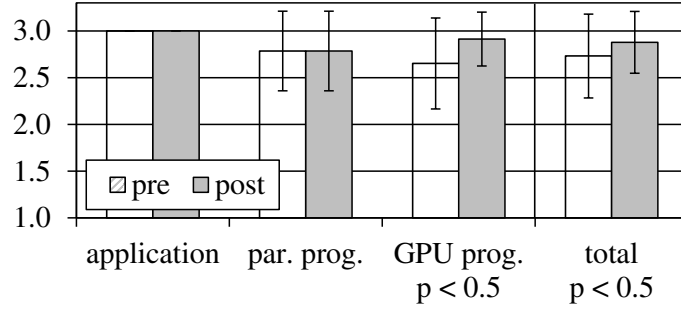


Figure 13.1.: Means of pre-KS and post-KS results per knowledge question group based on developer conducting *ZFS* GPU tuning. Error bars indicate standard deviation.

Table 13.1.: Comparison of tracked effort data using *EffortLog* and manual diaries for porting and tuning *ZFS* with OpenACC for GPUs.

	first parallel: man. diary	tuned parallel: <i>EffortLog</i>
effort reported [days]	11	64
effort estimated [days]	19	52
#entries/ logged hour	~0.34	~1.55
min interval [min]	10	1
max interval [min]	780	251
mean interval [min]	178.4	40.3
#milestones	1	56

Sect. 11.2), and look at the overall performance life-cycle. Furthermore, actual efforts are compared to estimated efforts using COCOMO II.

The first parallel version of *ZFS* has been developed in roughly 8 person-days and respective effort was logged using a manual developer diary. The second developer spent 40 days on the follow-up tuning and logged milestones with the electronic developer diary *EffortLog*. Since both developers do not belong to the domain scientists team, they had to setup the application in their cluster environment including various library installations. The first developer reported 2.5 days for that, while the second developer tracked 24 person-days because he encountered a major update of the operating system during setup that caused incompatibilities in certain library versions and required many re-installations and tests. Comparing characteristics of manual and electronic diary logs in Tab. 13.1, it can be seen that the accuracy, i.e., the number of entries per hour, is much higher with *EffortLog* compared to manual reports. In addition, the maximum interval between entries is ~4 hours with *EffortLog*, and log categories reveal 13 hours of breaks (excluded from development effort) and 9 hours of thinking time. Instead, manual diary

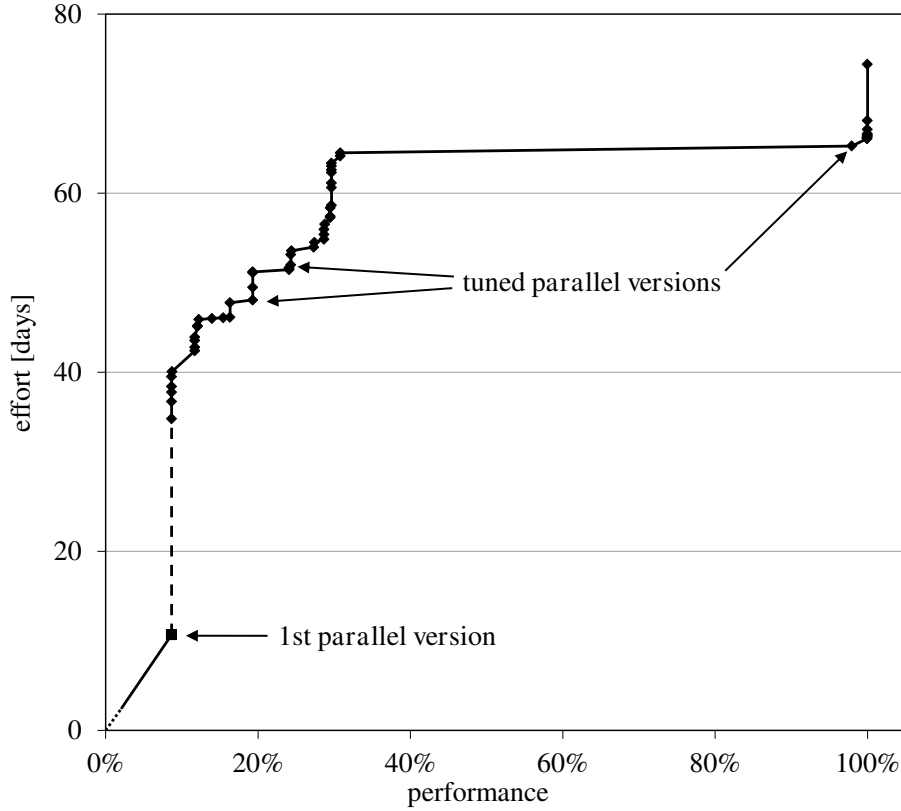


Figure 13.2.: Performance life-cycle of *ZFS* with respect to GPU development activities. Dashed lines represent setup efforts.

entries are apart for maximal 13 hours and do not explicitly list break or thinking time. Hence, the usage of *EffortLog* encourages to also log time spans that do not directly belong to development in front of the computer. Furthermore, using *EffortLog*, the second developer reported 56 performance-related milestones, while the manual diary only shows 1 milestone. With these milestones, the performance life-cycle of *ZFS* can be examined. Figure 13.2 illustrates this life-cycle where data points represent tracked milestones and dashed lines symbolize the setup times. The corresponding tuning steps are further described in [Mil16].

With respect to COCOMO’s applicability to HPC projects, Miller and I compare reported and estimated efforts with COCOMO’s Post Architecture Model in [MWSL⁺17]. As evident in Tab. 13.1, COCOMO highly overestimates effort spent for creating the first parallel OpenACC version of *ZFS* (relative error is 80 %). Contrarily, it underestimates required effort by 20 % for GPU tuning activities and supports our expectation that HPC activities require more effort than regular software development. Reasons for differences between reported and estimated efforts are discussed in Sect. 9.2.1.

14. Total Costs

The next step covers the estimation of total costs with respect to Equ. (7.1). Aggregated expenses serve as denominator in the productivity index whose purpose is to compare different hardware and application setups. Quantified values are based on experiences from the compute cluster installed at RWTH Aachen University.

14.1. System-Dependent Components

For HPC setup comparison, I focus on two types of compute nodes: standard two-socket Intel Sandy Bridge servers (SNB) and GPU-based SNB nodes with one NVIDIA Kepler K20 GPU (K20) as described in Tab. A.4. Hardware purchase and maintenance costs are real values taken from procurement of these systems in 2013. Since, the original compute nodes acquired by RWTH Aachen University comprise two Intel sockets and two NVIDIA K20 GPUs, expenses and power consumptions for SNB and K20 nodes are derived by cost breakdowns and subtraction by GPU idle power. Furthermore, I investigate single-node executions of *ZFS*, so that the network component is not important.

Other system-dependent costs are analogously set up to previously-described use cases (compare Appendix A.2.2) with compiler and software costs set to 50,000 €. All power measurements are done with an LMG450 power meter from ZES. Staff salaries are based on recommendations of the DFG for 2017 [Ger17] and amount to 301.43 € per person-day.

14.2. Application-Dependent Components

Since this business case is based on a single-application perspective, application-dependent costs cover expenses for *ZFS* development and deployment only.

Development effort for porting and tuning of *ZFS* amounts to 75 person-days in the OpenACC case as quantified in Sect. 13.2. For the original base MPI version of the code, I do not consider any effort costs as the focus is on the cost effectiveness of moving *ZFS* to an accelerator-type hardware. Moreover, maintenance effort is ignored for simplicity.

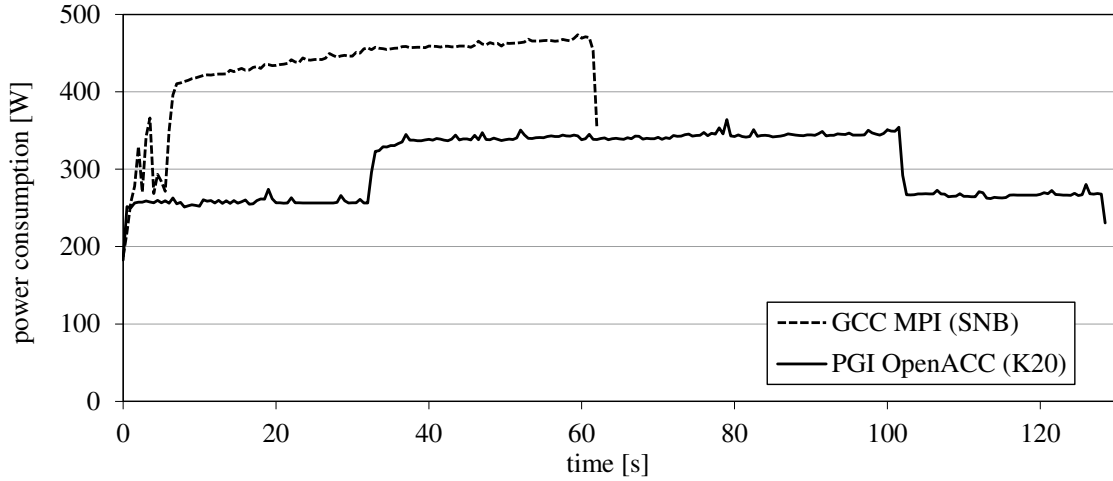


Figure 14.1.: Trace of power consumption of original MPI version (compiled with GCC) and tuned OpenACC version of *ZFS*.

For application-dependent power consumptions, I use the model introduced in Equ. (7.3) that distinguishes between average consumptions in serial and parallel parts of the code. For that, power traces over the application's runtime are created at a sample rate of 0.5s. The aggregated consumption measured at the nodes' two power supplies is illustrated in Fig. 14.1. Here, the MPI code version executed on SNB consumes on average 253.06 W in the initialization part of the code and 412.66 W in the parallel DG solver. As evident from the peaks in the trace (dashed curve in Fig. 14.1), the initialization part is not completely serially executed. Nevertheless, this simplified power model still takes the average watt number for TCO computations. The power trace of the OpenACC version (pure line) shows a plateau with an average of 323.83 W that represents the parallel execution (of the DG's main loop) on the GPU. The (actually-serial) initialization and finalization time is much higher due to some error checks that are not well translated by the PGI compiler. Further information on this issue is presented in Sect. 15.1.

15. Productivity

Putting it all together, I use the productivity model from Equ. (4.3) to compare MPI and OpenACC versions of *ZFS* and determine a reasonable system lifetime. I focus on an ex-post analysis with quantifications from the previous sections. In addition, I provide an outlook on transferring known values to a future hardware in an ex-ante analysis with respect to procurement.

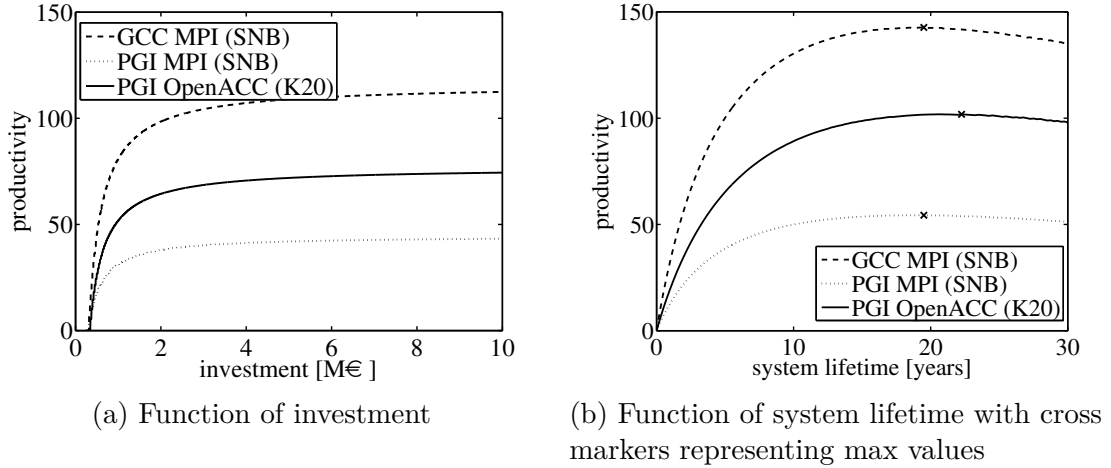
15.1. Ex-Post Analysis

The HPC setups under investigation cover the MPI version of *ZFS* running on SNB, and the OpenACC version leveraging a single Kepler GPU. For the MPI versions, I differentiate between results obtained with compilations from GCC (best effort) and PGI (as reference for PGI's OpenACC implementation).

For the computation of the productivity's numerator, i.e., the number of application runs, *ZFS* runtime has been measured using GCC 4.8.5 and PGI 16.4 with OpenMPI 1.10.2 on SNB, and PGI 16.4 with CUDA 7.5 on K20. Performance results given in Tab. 15.1 are averaged values over 10 runs. Since especially MPI runs show noticeable deviations, mean values are appropriate to represent the expected runtime value. Furthermore, the overall runtime of *ZFS* includes (at least) two serial L^∞ -norm computations that serve as error checks. These parts constitute a major runtime fraction for PGI-translated code in the small test case, since the PGI compiler cannot correctly apply inlining (amongst others) to these

Table 15.1.: Application-dependent productivity parameters of *ZFS*.

Code version	System	Programming model	Effort [days]	Kernel runtime [s]
CPU base (GCC)	SNB	MPI	-	61.8
CPU base (PGI)	SNB	MPI	-	165.1
first parallel	K20	OpenACC	11	869.7
tuned parallel	K20	OpenACC	64	69.5

Figure 15.1.: Productivity of *ZFS* setups.

code parts. Since this runtime fraction diminishes for real-sized data sets, I exclude these two error checks from *ZFS* total runtime. A further discussion on PGI’s impact on runtime can be found in [Mil16]. As evident from Tab. 15.1, the PGI compiler also introduces penalties into code translation for the MPI version. While the first parallel OpenACC version is considerably slower than both MPI versions, the tuned GPU version roughly competes with the best-effort CPU runtime. Miller [Mil16] provides a performance analysis for differences in CPU and GPU versions. In the following, I focus on the tuned GPU version with an aggregated development effort of 75 person-days.

Other assumptions for productivity calculations cover a constant quality weighting factor $q = 1$ and a system availability of $\alpha = 0.8$. Due to the single-node code version, it holds that $n_{\text{scale}} = n_i = 1$. The default system lifetime is set to 6 years and the default investment to 5 Mio. €.

Productivity results are shown in Fig. 15.1. As expected, the OpenACC version cannot compete with the best-effort MPI version in productivity due to its additional effort costs and higher hardware expenses (see Fig. 15.1a). However, its performance can compensate for that with respect to the PGI MPI version running on SNB. Keeping the investment fixed and varying the system’s lifetime (Fig. 15.1b) reveals that HPC managers do not have to worry about typical funding periods of 6 years since the productivity rises until roughly 20 years. Note that this only holds under the assumption that vendor’s would be willing to provide long-term maintenance contracts at same conditions (see Sect. 7.2). However, since components will most-likely break down after a couple of years, a reasonable system lifetime will depend on these contractual maintenance periods.

15.2. Ex-Ante Analysis

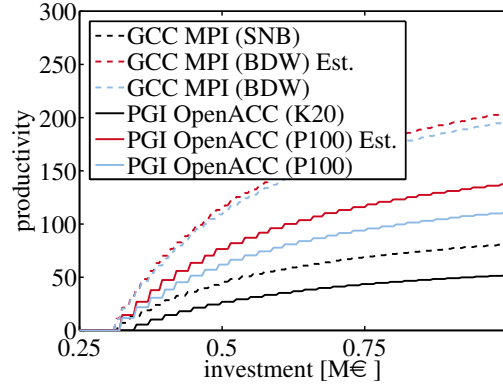
Moving from the productivity analysis of experimentally-gathered data to an ex-ante procurement analysis, I estimate productivity of an RWTH HPC setup with novel hardware. As CPU standard server, I assume Intel Broadwell nodes (see BDW in Tab. A.4), whereas a single NVIDIA Pascal GPU (P100 in Tab. A.4) attached to a BDW host serves as up-to-date GPU-based node.

Most of the system-dependent components are kept as in the previous analysis, i.e., for example, the staff costs for administration and the software license costs since they are restricted by available budgets. The hardware purchase costs are taken from official vendor's machine offers in 2016. Since the offer covered only dual GPU nodes, I reduced the purchase price by the cost of one Pascal GPU (roughly 8,000 €). Since the annual infrastructure/ building costs are dependent on the maximum energy consumption per node (in this quantification), I model the latter by vendor information on TDP of the CPUs and GPUs.

For the application-dependent *ZFS* costs, I omit expenses for development efforts since I ignore additional tuning for now. Furthermore, a previous MPI code version should work on Broadwell nodes without extra effort. For the GPU code version, I assume the same as OpenACC leaves the responsibility to create code for new architectures to the compiler. For estimations on power consumptions for serial and parallel code portions, I use respective power measurements from another application as replacement. However, these values could also be provided by the vendor. I create a basic performance model to predict kernel runtimes while keeping the runtimes of the serial code portion fixed. I assume that *ZFS* is memory-bound for most code parts and, therefore, I investigate sustainable memory bandwidths on previous and future hardware systems. The Stream benchmark [McC95] reveals a factor of 1.6 moving from SNB to BDW (see Tab. 15.2). Going from K20 to P100, the GPU-Stream [DPMMS16] values suggest a factor of 3.1. Kernel runtimes are predicted applying the respective factors. For productivity comparison and evaluation of possible effects of simple models, another pro-

Table 15.2.: Performance estimations for *ZFS* based on Stream bandwidth (BW) measurements, and actual kernel runtime measurements.

HW	Stream BW [GB/s]	Factor	Estimated <i>ZFS</i> kernel runtime [s]	Measured <i>ZFS</i> kernel runtime [s]	Factor
SNB	74			61.8	
BDW	120	1.6	38.6	39.1	1.58
K20	180			69.5	
P100	550	3.1	22.4	31.3	2.22

Figure 15.2.: Estimation of productivity of *ZFS* setups.

ductivity setup replaces some estimated values by actual measurements in terms of parallel and serial runtimes and power measurements for the node’s maximum power consumption approximated by running Linpack.

A comparison of corresponding productivity results can be found in Fig. 15.2 with variation over investment amounts. Here, black curves represent the productivity from the ex-post analysis, red lines the estimated productivity on novel hardware, and blue-colored functions the productivity substantiated by measurements. Moving from the previous to next generation hardware, productivity increases as expected. Especially, runtime has been decreased while roughly keeping the power consumption and spending no further development efforts. Remarkably, P100-GPU productivity (pure lines in Fig. 15.2) is below BDW-CPU productivity (dashed curves) although GPU performance improved over CPU performance. Investigation of components shows that the main reason for that are much higher purchase costs for GPU-based nodes that cannot be compensated by reduced runtime. To illustrate effects from prediction errors, I compare productivity stemming from estimated and measured components (red vs. blue lines in Fig. 15.2). As seen in the sensitivity analysis in Sect. 4.6, productivity variance is sensitive to errors in the kernel runtime prediction. Since the simple bandwidth-based performance model for *ZFS* kernels estimates the runtime behavior for standard servers very well (factor 1.6 instead of 1.58), corresponding productivity results are close with 284 at 10 Mio. € for the estimation-driven productivity and 272 for the measurement-driven productivity. Contrarily, this bandwidth model approach is less suitable for GPU-based performance. It delivers a factor 3.1 instead of 2.22 and, thus, shows a divergence of 37 productivity points at 10 Mio. €. While a detailed analysis of the performance model’s suitability is out of the scope of this work, one possible reason for the performance difference comes from Miller [Mil16] who suggests that GPU performance is not only bound by memory bandwidth but also latency that cannot be well hidden for the *ZFS* data structures on the GPU.

15.3. Future Directions

Drawing conclusions from porting *ZFS* to GPUs with an accompanied PLC evaluation, I present benefits of these HPC activities for the Institute of Aerodynamics and lessons learned for future productivity research projects.

Enabling *ZFS* for GPU usage was conducted on a dedicated software branch. The project's success initiated the inclusion of the OpenACC parallelization into the production code. However, due to some major issues with PGI's implementation in combination with the institutes's production environment, these activities are deferred at the moment. Nevertheless, the collaboration is ongoing in terms of GPU tuning for the multi-node MPI code version since the Institute of Aerodynamics is interested in being able to explore more HPC cluster types to run more scientific simulations. This is fostered by the prospect of a well-performing OpenACC implementation of GCC in near future.

In terms of productivity research, this case study shows that *EffortLog* is applicable during real-world code development over a longer time period and that it delivers valuable data on effort-performance pairs. Thus, this business case encourages the application of *EffortLog* and other effort estimation concepts to future real-world HPC use cases.

16. Conclusion

In this thesis, I presented methodologies and models with focus on informed HPC system procurement in Germany. My first contribution is a productivity figure of merit of HPC centers that covers the value of scientific simulations over total ownership costs of multi-job HPC environments. The comprehensive investigation of HPC total costs and their quantification and predictability capabilities leads to my methodology of effort estimation for HPC development of scientific applications. It includes the invention of a performance life-cycle that describes the effort-performance relationship, as well as the identification and quantification of key drivers on HPC software development effort. With online material and tools, I foster the challenging collection of data sets in human-subject research.

In Part A of this thesis, I discussed shortcomings of previous HPC productivity approaches such as limited quantification and predictability abilities of model parameters and simplification of multiple-application environments. In my productivity model, I put the focus on these characteristics by introducing an aggregated number of all application runs as value metric and showed predictability of almost all model parameters. The resulting metric models productivity as function of system size, i.e., the number of cluster compute nodes, and the system's lifetime. The value metric as numerator of productivity covers the sum of application runs during the system lifetime and considers the applications' runtimes, their scaling characteristics in terms of quality weighting factors, as well as their system capacity shares. As denominator of productivity, I modeled total ownership costs of the HPC center as shown in Part B. Furthermore, I illustrated the value of my productivity metric in numerous use cases, e.g., the comparison of different hardware setups, the determination of a reasonable system lifetime or the comparison of single- and two-phase procurements. I also analyzed the model's sensitivity towards errors in its parameters. With only few sensitive parameters that can be well estimated, I showed that my approach is robust within the given conditions.

Part B solidifies my model of total costs of ownership for HPC centers. It distinguishes one-time and annual costs, as well as node-based and node-type-based parameters. It incorporates components such as purchase costs of compute nodes and infrastructure, administration effort for environment installation and maintenance, power consumption, software licenses, and development effort for parallelizing, porting or tuning applications. Furthermore, I covered particularities

for job-mix setups and assumptions on power consumption during system unavailabilities. A discussion on quantification and predictability of all TCO parameters concludes this part.

As one TCO component with increasing importance in procurement decisions, development effort needed for HPC activities is considerably challenging to quantify and predict, and covered in Part C. First, I found that software complexity metrics, e.g., lines of code or function points, from traditional software engineering do not appropriately cover complexity of HPC activities and, thus, cannot serve as quantifier for HPC effort. Instead, I comprehensively defined HPC development time as suitable quantifier. Similarly, the application of the popular software cost estimation model COCOMO II to some HPC projects reveals that it is not appropriately applicable to estimate HPC development time. As consequence, I introduced a methodology that covers the estimation of software development effort in HPC. It focuses on — what I called — the *performance life-cycle* that models development time as function of performance and in dependence of numerous other parameters such as experiences of the developer, the parallel programming model or available tool support. I illustrated the interpretation of the 80-20 rule as possible performance life-cycle and compared it to step-wise functions arising in real world. To reveal the course of function and their parameter dependencies, my methodology follows the identification of important impact factors, i.e., key drivers. For that, I used ranking surveys and early results from 44 participants show that pre-knowledge on the hardware architecture, parallel programming model and numerical algorithm is a strong key driver for them, as well as the kind of code work and the parallel programming model itself. As next step, my methodology comprises the quantification of key drivers to enable a regression analysis of my effort estimation model. To quantify the effect of pre-knowledge, I introduced the concept of knowledge surveys borrowed from educational assessment. Since they have been mainly used in classes for natural sciences, I created various knowledge surveys with respect to high-performance computing. I showed as proof of concept through their application in GPU hackathons and student software labs that pre and post event surveys yield significant differences in confidence ratings. Covering another key driver, I also investigated the impact of the parallel programming and numerical algorithm on HPC effort and presented ideas and methods to quantify their impact. Finally, my methodology includes the data collection for the performance life-cycle quantification, i.e., obtaining effort-performance pairs from software developers in real-world HPC projects. Since self-reported data in form of developer diaries tends to be very coarse grained, the electronic developer diary *EffortLog* was established to support the process of effort and performance tracking. In early results, I showed that *EffortLog* enables more accurate tracking than manual diaries. Additionally, I spotlighted numerous challenges that arise when collecting data from human subjects. I emphasized the required community effort to gather reliable data and help jump start it with online material.

Assembling introduced methods, I made the business case in Part D using an aeroacoustics simulation application. I gathered knowledge surveys and effort-performance logs to illustrate a real-world performance life-cycle, and quantified TCO and productivity components. I exemplified the transfer from ex-post to ex-ante procurement analysis, and the predictive power of my productivity model.

Concluding, based on experiences from RWTH Aachen University, I put methodologies and models in place to foster informed decisions for HPC procurements. Productivity and TCO models are set up to cover a broad spectrum of use cases in German university HPC centers that recognize the increasing importance of brainware for HPC deployments. In addition, my research contributed to the call for tenders for the RWTH Compute Cluster CLAIX in 2016. While the corresponding tender documents abstracted from fixed costs that are the same for all vendors, it incorporated purchase and energy costs for a real-world RWTH job mix.

Future Work

As foundation for future estimations and evaluations in HPC procurement processes, I will continue to establish an effort estimation model suitable for HPC. Therefore, I will substantiate my methodology by collecting further data sets with respect to ranking surveys, knowledge surveys and effort-performance logs, and encourage external collection by participation in hackathons or other events, and motivation of managers and researchers in talks and through online material and communication. Furthermore, I will extend my methodology for quantification of further key drivers and describe the performance life-cycle by regression analysis.

Besides development effort, the discussion on predictability of TCO components revealed that application maintenance effort needs further investigations. If infeasible to interpret maintenance effort as a part of my concept for software development effort estimation, novel methodologies must be established. Additionally, other cost components like the procurement project itself or the commissioning of the cluster, have not been accounted so far. For future cluster acquisitions, I will motivate to collect corresponding data to expose the significance of their impact.

All previous aspects naturally affect the productivity model as well, since it incorporates TCO and brainware components. Currently, it simplifies some effects with respect to its (life)time parameter which I will further investigate and include into model extensions if needed: First, it assumes costs to arise proportionally each year instead of expressing parameters as integral over time. Second, it does not overlap development time (or effort schedules) with the system's lifetime causing emphasis on performance. Third, it excludes pre-production phases and, thus, does not aggregate corresponding simulation results to the value of the cluster. Finally, I will enhance my groundwork on tool support for productivity evaluations to enable HPC managers and developers to easily use and adapt the model.

Bibliography

- [AAD⁺10] Amy Apon, Stanley Ahalt, Vijay Dantuluri, Constantin Gurdgiev, Moez Limayem, Linh Ngo, and Michael Stealey. High Performance Computing Instrumentation and Research Productivity in U.S. Universities. *Journal of Information Technology Impact*, 10(2):87–98, 2010.
- [AB07] E. Aranha and P. Borba. Test Effort Estimation Models Based on Test Specifications. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, pages 67–71, 2007.
- [ABC⁺06] Krste Asanović, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.
- [AG83] Allan J. Albrecht and J. E. Gaffney. Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation. *IEEE Transactions on Software Engineering*, SE-9(6):639–648, 1983.
- [AHL⁺11] Saman Amarasinghe, Mary Hall, Richard Lethin, Keshav Pingali, Dan Quinlan, Vivek Sarkar, John Shalf, Robert Lucas, Katherine Yelick, and Pavan Balanji. Exascale Programming Challenges. In *Proceedings of the Workshop on Exascale Programming Challenges, Marina del Rey, CA, USA. US Department of Energy, Office of Science, Office of Advanced Scientific Computing Research*, 2011.
- [Alb79] Allan J. Albrecht. Measuring Application Development Productivity. *Proceedings of the Joint SHARE/GUIDE/IBM Application Development Symposium*, pages 83–92, 1979.
- [ANPW15] Amy Apon, Linh Ngo, Michael E. Payne, and Paul W. Wilson. Assessing the Effect of High Performance Computing Capabilities on Academic Research Output. *Empirical Economics*, 48(1):283–312, 2015.

- [BAB⁺00a] Barry Boehm, Chris Abts, A. Winsor Brown, Sunita Chulani, Brad Clark, Ellis Horowitz, Ray Madachy, Don Reifer, and Bert Steece. COCOMO II - Model Definition Manual, Version 2.1. Technical report, University of Southern California, 2000.
- [BAB⁺00b] Barry Boehm, Chris Abts, A. Winsor Brown, Sunita Chulani, Brad Clark, Ellis Horowitz, Ray Madachy, Don Reifer, and Bert Steece. *Software Cost Estimation with Cocomo II*. Prentice Hall PTR, 2000.
- [BAC00] Barry Boehm, Chris Abts, and Sunita Chulani. Software development cost estimation approaches - A survey. *Annals of Software Engineering*, 10(1):177–205, 2000.
- [BaMI12] Christian Bischof, Dieter an Mey, and Christian Iwainsky. Brainware for green HPC. *Computer Science - Research and Development*, 27(4):227–233, 2012.
- [BBD⁺13] Richard F Barrett, Shekhar Borkar, Sudip S Dosanjh, Simon D Hammond, Michael A Heroux, X Sharon Hu, Justin Luitjens, Steven G Parker, John Shalf, and Li Tang. *On the Role of Co-design in High Performance Computing*, volume 24 of *Advances in Parallel Computing*, pages 141–155. IOS Press, 2013.
- [BBH05] Nancy Bowers, Maureen Brandon, and Cynthia D. Hill. The Use of a Knowledge Survey as an Indicator of Student Learning in an Introductory Biology Course. *Cell Biology Education*, 4(4):311–322, 2005.
- [BBR08] M. Bücker, R. Beucker, and A. Rupp. Parallel Minimum p -Norm Solution of the Neuromagnetic Inverse Problem for Realistic Signals Using Exact Hessian-Vector Products. *SIAM Journal on Scientific Computing*, 30(6):2905–2921, 2008.
- [BCH⁺95] Barry Boehm, Bradford Clark, Ellis Horowitz, Chris Westland, Ray Madachy, and Richard Selby. Cost models for future software life cycle processes: COCOMO 2.0. *Annals of Software Engineering*, 1(1):57–94, 1995.
- [BCH13] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. *The Data-center as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2nd edition, 2013.
- [BD08] Manfred Bundschuh and Carol Dekkers. *Product- and Process- Metrics*, pages 207–239. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

-
- [BIP08] BIPM, IEC and IFCC, ILAC and ISO, IUPAC and IUPAP, OIML. Evaluation of measurement data—Guide to the expression of uncertainty in measurement, JCGM 100:2008, GUM 1995 with minor corrections. *Joint Committee for Guides in Metrology*, 2008.
- [Bir13] Birst Inc. Comparing the Total Cost of Ownership of Business Intelligence Solutions - How Cloud BI Can Reduce TCO by 70% versus Traditional and Open Source BI. Technical report, Birst Inc., 2013.
- [Blo56] Benjamin Samuel Bloom. *Taxonomy of educational objectives: The classification of educational goals: Cognitive Domain*. Longman, 1956.
- [Boe81] Barry Boehm. *Software Engineering Economics*. Prentice Hall PTR, 1981.
- [Boe96] Barry Boehm. Anchoring the Software Process. *IEEE Software*, 13(4):73–82, 1996.
- [Bru17] Rich Brueckner. How GPU Hackathons Bring HPC to More Users. insideHPC, <https://insidehpc.com/2017/03/gpu-hackathons-bring-hpc-users>, Accessed June 2017, 2017.
- [BS03] David Bailey and Allan Snaveley. Performance Modeling, Metrics, and Specifications. In Daniel A. Reed, editor, *Workshop on the Roadmap for the Revitalization of High-End Computing*, pages 59–67. Computing Research Association, 2003.
- [BV11] Priscilla Bell and David Volckmann. Knowledge Surveys in General Chemistry: Confidence, Overconfidence, and Performance. *Journal of Chemical Education*, 88(11):1469–1476, 2011.
- [BZ09] Victor R. Basili and Marvin V. Zelowitz. The Use of Empirical Studies in the Development of High End Computing Applications. Technical Report AFRL-RI-RS-TR-2009-278, University of Maryland, 2009.
- [CBS99] S. Chulani, B. Boehm, and B. Steece. Bayesian analysis of empirical software engineering cost models. *IEEE Transactions on Software Engineering*, 25(4):573–583, 1999.
- [CDS85] S. D. Conte, H. E. Dunsmore, and V. Y. Shen. Software Effort Estimation and Productivity. *Advances in Computers*, 24:1–60, 1985.
- [CDS00] B. L. Chamberlain, S. J. Deitz, and L. Snyder. A Comparative Study of the NAS MG Benchmark across Parallel Languages and Architectures. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, pages 46–46, 2000.

- [CES12] Iris Christadler, Giovanni Erbacher, and Alan D. Simpson. *Performance and Productivity of New Programming Languages*, pages 24–35. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
- [CG12] Jon M. Clauss and C. Kevin Geedey. Knowledge surveys: Students ability to self-assess. *Journal of the Scholarship of Teaching and Learning*, 10(2):14–24, 2012.
- [Cha16] Chair for High Performance Computing, RWTH Aachen University. Lecture on Programming. <http://www.hpc.rwth-aachen.de/teaching/>, 2015/2016.
- [Che88] W. Bruce Chew. No-Nonsense Guide to Measuring Productivity. *Harvard Business Review*, 66(1):110, 1988.
- [CHS10] A. G. Carlyle, S. L. Harrell, and P. M. Smith. Cost-Effective HPC: The Community or the Cloud? In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, pages 169–176, 2010.
- [CK10] Stephanie Riegg Cellini and James Edwin Kee. *Cost-effectiveness and cost-benefit analysis*, pages 493–530. John Wiley & Sons, 3 edition, 2010.
- [CL13] Hélène Coullon and Sébastien Limet. Algorithmic skeleton library for scientific simulations: SkelGIS. In *International Conference on High Performance Computing and Simulation (HPCS)*, pages 429–436, 2013.
- [Col91] Murray I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1991.
- [Col16] Toni Collis. Women in HPC. <http://www.womeninhpc.org>, Accessed October 2016.
- [Cou14] Council on Competitiveness. Solve. The Exascale Effect: the Benefits of Supercomputing Investment for U.S. Industry. Technical report, 2014.
- [CSKaM12] T. Cramer, D. Schmidl, M. Klemm, and D. an Mey. OpenMP Programming on Intel Xeon Phi Coprocessors: An Early Performance Comparison. In *Proceedings of the Many-core Applications Research Community (MARC) Symposium at RWTH Aachen University*, pages 38–44, 2012.
- [CYZEG04] F. Cantonnet, Y. Yao, M. Zahran, and T. El-Ghazawi. Productivity analysis of the UPC language. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, page 254, 2004.

-
- [Dam14] Alesja Dammer. Parallel Design Patterns in Comparison on Intel Xeon Phi and NVIDIA GPUs. Master thesis, Aachen University of Applied Sciences, 2014.
- [DaMW⁺13] Fan Ding, Dieter an Mey, Sandra Wienke, Ruisheng Zhang, and Lian Li. An HPC Application Deployment Model on Azure Cloud for SMEs. In Frédéric Desprez, Donald Ferguson, Ethan Hadar, Frank Leymann, Matthias Jarke, and Markus Helfert, editors, *Proceedings of the 3rd International Conference on Cloud Computing and Services Science (CLOSER 2013)*, pages 253–259. SCITEPRESS, 2013.
- [DaMW⁺14] Fan Ding, Dieter an Mey, Sandra Wienke, Ruisheng Zhang, and Lian Li. A Study on Today’s Cloud Environments for HPC Applications. In Markus Helfert, Frédéric Desprez, Donald Ferguson, and Frank Leymann, editors, *Cloud Computing and Services Science*, volume 453 of *Communications in Computer and Information Science*, pages 114–127. Springer International Publishing, 2014.
- [DD04] Jack Dongarra and Bronis R. De Supinski, editors. *International Journal of High Performance Computing Applications*, volume 18(4). Sage Publications, Thousand Oaks, CA, USA, 2004.
- [DGH⁺08] Jack Dongarra, Robert Graybill, William Harrod, Robert Lucas, Ewing Lusk, Piotr Luszczek, Janice McMahon, Allan Snaveley, Jeffrey Vetter, Katherine Yelick, Sadaf Alam, Roy Campbell, Laura Carrington, Tzu-Yi Chen, Omid Khalili, Jeremy Meredith, and Mustafa Tikir. DARPA’s HPCS Program: History, Models, Tools, Languages. In Marvin V. Zelkowitz, editor, *Advances in COMPUTERS High Performance Computing*, volume 72 of *Advances in Computers*, pages 1–100. Elsevier, 2008.
- [DH11] Timothy A. Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1:1–1:25, 2011.
- [DH13] Jack Dongarra and Michael A. Heroux. Toward a New Metric for Ranking High Performance Computing Systems. Technical report, Sandia National Laboratories, 2013.
- [DOE14] DOE ASCAC Subcommittee. Top Ten Exascale Research Challenges. Technical report, U.S. Department of Energy (DOE), 2014.
- [DOE17a] U.S. Department of Energy (DOE). The Challenges of Exascale. <https://science.energy.gov/ascr/research/scidac/exascale-challenges/>, Accessed January 2017.

- [DOE17b] U.S. Department of Energy (DOE). Co-Design. <https://science.energy.gov/ascr/research/scidac/co-design/>, Accessed October 2017.
- [DPMMS16] Tom Deakin, James Price, Matt Martineau, and Simon McIntosh-Smith. *GPU-STREAM v2.0: Benchmarking the Achievable Memory Bandwidth of Many-Core Processors Across Diverse Parallel Programming Models*, pages 489–507. Springer International Publishing, 2016.
- [DWZ15] Fan Ding, Sandra Wienke, and Ruisheng Zhang. Dynamic MPI parallel task scheduling based on a master-worker pattern in cloud computing. *International Journal of Autonomous and Adaptive Communications Systems*, 8(4):424–438, 2015.
- [EA16] Stephen J. Ezell and Robert D. Atkinson. The Vital Importance of High-Performance Computing to U.S. Competitiveness. Technical report, Information Technology & Innovation Foundation (ITIF), 2016.
- [ECSC09] Giovanni Erbacci, Carlo Cavazzoni, Filippo Spiga, and Iris Christadler. Report on petascale software libraries and programming models. Technical Report D6.6, Partnership for Advanced Computing in Europe (PRACE), 2009.
- [ESEG⁺06] Kemal Ebcioglu, Vivek Sarkar, Tarek El-Ghazawi, John Urbanic, and P Center. An experiment in measuring the productivity of three parallel programming languages. In *Workshop on Productivity and Performance in High-End Computing (P-PHEC)*, pages 30–36, 2006.
- [Eur14] European Commission - Community Research and Development Information Service (CORDIS). Guide to Financial Issues relating to FP7 Indirect Actions. http://ec.europa.eu/research/participants/data/ref/fp7/89556/financial_guidelines_en.pdf, Accessed March 2017, 2014.
- [Eur17] European Commission. Horizon 2020–Work Programme 2016 - 2017, 2. Future and Emerging Technologies. Technical report, 2017.
- [FB14] Norman Fenton and James Bieman. *Software Metrics: A Rigorous and Practical Approach*. CRC Press, third edition edition, 2014.
- [FBHK05] Andrew Funk, Victor Basili, Lorin Hochstein, and Jeremy Kepner. Application of a development time productivity metric to parallel software development. *Proceedings of the Second International Workshop on Software engineering for high performance computing system applications (SE-HPCS '05)*, pages 8–12, 2005.

-
- [FBHK06] Andrew Funk, Victor Basili, Lorin Hochstein, and Jeremy Kepner. Analysis of Parallel Software Development Using the Relative Development Time Productivity Metric. *CTWatch Quarterly*, 2(4A):46–51, 2006.
- [Fel17] Michael Feldman. China Will Deploy Exascale Prototype This Year. Top500, <https://www.top500.org/news/china-will-deploy-exascale-prototype-this-year/>, Accessed January 2017.
- [FF14] Ian Farrance and Robert Frenkel. Uncertainty in measurement: a review of Monte Carlo simulation using Microsoft Excel for the calculation of uncertainties through functional relationships, including uncertainties in empirically derived constants. *The Clinical Biochemist Reviews*, 35(1):37–61, 2014.
- [FGJ⁺04a] S. Faulk, J. Gustafson, P. Johnson, A. Porter, W. Tichy, and L. Votta. Toward accurate HPC productivity measurement. In Philip M. Johnson, editor, *Proceedings of the First International Workshop on Software Engineering for High Performance Computing System Applications*, pages 42–46, 2004.
- [FGJ⁺04b] S. Faulk, J. Gustafson, P. Johnson, A. Porter, W. Tichy, L. Votta, and F. U. Edu. Measuring high performance computing productivity. *International Journal of High Performance Computing Applications*, 18(4):459–473, 2004.
- [Fie13] Andy Field. *Discovering Statistics Using IBM SPSS Statistics*. Sage Publications Ltd., 4th edition, 2013.
- [FKB05] Andrew Funk, Jeremy Kepner, and Victor Basili. A Relative Development Time Productivity Metric for HPC Systems. In *Ninth Annual Workshop on High Performance Embedded Computing*, pages 20–22, 2005.
- [FWW14] Lacey Favazzo, John D. Willford, and Rachel M. Watson. Correlating Student Knowledge and Confidence Using a Graded Knowledge Survey to Assess Student Learning in a General Microbiology Classroom. *Journal of Microbiology & Biology Education*, 15(2):251–258, 2014.
- [Gau16] Gauß-Allianz e.V. Strategische Aufgaben der Gauß-Allianz in einem nationalen HPC-Konzept. Technical report, 2016.
- [Gau17a] Gauß-Allianz e.V. Gauß-Allianz. <https://gauss-allianz.de/en>, Accessed April 2017.

- [Gau17b] Gauss Centre for Supercomputing. Gauss Centre for Supercomputing (GCS). www.gauss-centre.eu, Accessed April 2017.
- [Gen16] Wolfgang Gentzsch. A Total Cost Analysis for Manufacturers of In-house Computing Resources and Cloud Computing. Technical report, UberCloud, 2016.
- [Ger13] German Science Foundation (DFG). DFG Personnel Rates for 2013. http://www.dfg.de/formulare/60_12_-2013-/60_12_en.pdf, Accessed March 2017, 2013.
- [Ger16] German Science Foundation (DFG). Informationsverarbeitung an Hochschulen - Organisation, Dienste und Systeme: Stellungnahme der Kommission für IT-Infrastruktur für 2016-2020. Technical report, 2016.
- [Ger17] German Science Foundation (DFG). DFG Personnel Rates for 2017. http://www.dfg.de/formulare/60_12/60_12_en.pdf, Accessed March 2017, 2017.
- [GG15] Jens Henrik Göbbert and Michael Gauding. psOpen. http://www.fz-juelich.de/ias/jsc/EN/Expertise/High-Q-Club/psOpen/_node.html, 2015.
- [GHLK⁺12] B. Grot, D. Hardy, P. Lotfi-Kamran, B. Falsafi, C. Nicopoulos, and Y. Sazeides. Optimizing Data-Center TCO with Scale-Out Processors. *IEEE Micro*, 32(5):52–63, 2012.
- [GKL⁺12] Richard Gluga, Judy Kay, Raymond Lister, Sabina Kleitman, and Tim Lever. Coming to terms with Bloom: an online tutorial for teachers of programming fundamentals. In *Proceedings of the Fourteenth Australasian Computing Education Conference*, volume 123, pages 147–156. Australian Computer Society, Inc., 2012.
- [GM11] A. Gupta and D. Milojevic. Evaluation of HPC Applications on Cloud. In *2011 Sixth Open Cirrus Summit*, pages 22–26, 2011.
- [GSS15] Corey Gough, Ian Steiner, and Winston Saunders. *Data Center Management*, pages 307–318. Apress, Berkeley, CA, 2015.
- [Gus04] J. Gustafson. Purpose-Based Benchmarks. *International Journal of High Performance Computing Applications*, 18(4):475–487, 2004.
- [GVL10] Horacio González-Vélez and Mario Leyton. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Software: Practice and Experience*, 40(12):1135–1160, 2010.
- [Hal77] Maurice Howard Halstead. *Elements of software science*. Operating and programming systems. Elsevier Science Inc., 1977.

-
- [Haq15] Riyaz Haque. HPC Development Effort Estimation Questionnaire: coMD & CnC. Internal questionnaires and interviews by Sandra Wienke, 2015.
- [Har17] Luke Harding. The node pole: inside Facebook’s Swedish hub near the Arctic Circle. <https://www.theguardian.com/technology/2015/sep/25/facebook-datacentre-lulea-sweden-node-pole>, Accessed May 2017.
- [HBVG08] Lorin Hochstein, Victor R. Basili, Uzi Vishkin, and John Gilbert. A pilot study to compare programming effort for two parallel programming models. *Journal of Systems and Software*, 81(11):1920–1930, 2008.
- [HBZ⁺05] Lorin Hochstein, Victor R. Basili, Marvin V. Zelkowitz, Jeffrey K. Hollingsworth, and Jeff Carver. Combining self-reported and automatic data to improve programming effort measurement. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 356–365, 1081762, 2005. ACM.
- [HCS⁺05] Lorin Hochstein, Jeff Carver, Forrest Shull, Sima Asgari, Victor Basili, Jeffrey K. Hollingsworth, and Marvin V. Zelkowitz. Parallel Programmer Productivity: A Case Study of Novice Parallel Programmers. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2005.
- [Hew17] Hewlett Packard Enterprise. HPE TCO and ROI Calculators. https://www.hpe.com/emea_europe/en/solutions/tco-calculators.html, Accessed May 2017.
- [HFA05] C. H. Hsu, W. C. Feng, and J. S. Archuleta. Towards Efficient Supercomputing: A Quest for the Right Metric. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS’05)*, 2005.
- [HKS⁺13] D. Hardy, M. Kleanthous, I. Sideris, A. G. Saidi, E. Ozer, and Y. Sazeides. An Analytical Framework for Estimating TCO and Exploring Data Center Design Space. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 54–63, 2013.
- [HMS08] Daniel Hartmann, Matthias Meinke, and Wolfgang Schröder. An adaptive multilevel multigrid formulation for Cartesian hierarchical grid methods. *Computers & Fluids*, 37:1103–1125, 2008.

- [HNS⁺08] Lorin Hochstein, Taiga Nakamura, Forrest Shull, Nico Zazworka, Victor R. Basili, and Marvin V. Zelkowitz. *Chapter 5 An Environment for Conducting Families of Software Engineering Experiments*, volume Volume 74, pages 175–200. Elsevier, 2008.
- [HS16] Jürgen Hedderich and Lothar Sachs. *Angewandte Statistik: Methodensammlung mit R*. Springer Spektrum, 15th edition, 2016.
- [HSS11] Damien Hardy, Isidoros Sideris, Ali Saidi, and Yiannakis Sazeides. EETCO: A tool to Estimate and Explore the implications of data-center design choices on the TCO and the environmental impact. In *Workshop on Energy-efficient Computing for a Sustainable World*, 2011.
- [Hum95] Watts S. Humphrey. *A Discipline for Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., 1995.
- [HZH⁺05] Jeff Hollingsworth, Marvin Zelkowitz, Lorin Hochstein, Sima Asgari, Victor Basili, and Taiga Nakamura. Measuring Productivity on High Performance Computers. *Software Metrics, IEEE International Symposium on*, page 6, 2005.
- [Ind17] Indiana University. Project: Open XDMoD Value Analytics (XDMoD-VA). <https://kb.iu.edu/d/anxb>, Accessed February 2017.
- [ITC13] IT Center of RWTH Aachen University, and Mathe-dual e.V.. HPC Battle 2012/13 for MATSE apprentices in Aachen, Jülich and Cologne. <http://www.itc.rwth-aachen.de/cms/IT-Center/Lehre-Ausbildung/MATSE/Aktuelle-Meldungen/~fiko/MATSE-aus-Aachen-Sieger-des-HPC-Battle/>, 2013.
- [ITC16] IT Center of RWTH Aachen University. Integrative Hosting. <https://doc.itc.rwth-aachen.de/display/IH>, Accessed November 2016.
- [ITC17a] IT Center of RWTH Aachen University. Project-based Management of Resources of the RWTH Compute Cluster. <https://doc.itc.rwth-aachen.de/display/CC/Project-based+Management+of+Resources+of+the+RWTH+Compute+Cluster>, Accessed April 2017.
- [ITC17b] IT Center of RWTH Aachen University. aixCAVE at RWTH Aachen University. <http://www.itc.rwth-aachen.de/cms/IT-Center/Forschung-Projekte/Virtuelle-Realitaet/Infrastruktur/~fgqa/aixCAVE/?lidx=1>, Accessed May 2017.

-
- [JAR17a] JARA - Jülich Aachen Research Alliance. Applying for computing time. <http://www.jara.org/en/research/hpc/partition-jade/computing-time>, Accessed October 2017.
- [JAR17b] JARA - Jülich Aachen Research Alliance. CSG Parallel Efficiency. <http://www.jara.org/en/research/hpc/cross-sectional-groups/parallel-efficiency>, Accessed October 2017.
- [JAR17c] JARA - Jülich Aachen Research Alliance. JARA-HPC. <http://www.jara.org/de/forschung/jara-hpc>, Accessed October 2017.
- [JAR17d] JARA - Jülich Aachen Research Alliance. The JARA Partition. <http://www.jara.org/en/research/hpc/partition-jade>, Accessed October 2017.
- [JBC⁺14] Guido Juckeland, William Brantley, Sunita Chandrasekaran, Barbara Chapman, Shuai Che, Mathew Colgrove, Huiyu Feng, Alexander Grund, Robert Henschel, Wen-Mei W. Hwu, Huian Li, Matthias S. Müller, Wolfgang E. Nagel, Maxim Perminov, Pavel Shelepugin, Kevin Skadron, John Stratton, Alexey Titov, Ke Wang, Matthijs van Waveren, Brian Whitney, Sandra Wienke, Rengan Xu, and Kalyan Kumaran. SPEC ACCEL: A Standard Application Suite for Measuring Hardware Accelerator Performance. In *Proceedings of the 5th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer System (PMBS 2014)*, 2014.
- [JCD13] Earl C. Joseph, Steve Conway, and Chirag Dekate. Creating Economic Models Showing the Relationship Between Investments in HPC and the Resulting Financial ROI and Innovation? and How It Can Impact a Nation’s Competitiveness and Innovation. Technical report, International Data Corporation (IDC), 2013.
- [JCSM16] Earl C. Joseph, Steve Conway, Robert Sorensen, and Kevin Monroe. IDC HPC ROI Research Update: Economic Models For Financial ROI And Innovation From HPC Investments. DOE, Advanced Scientific Computing Advisory Committee, <https://science.energy.gov/ascr/ascac/meetings/201612>, Accessed January 2017, 2016.
- [JHA⁺03] P. M. Johnson, Kou Hongbing, J. Agustin, C. Chan, C. Moore, J. Miglani, Zhen Shenyan, and W. E. J. Doane. Beyond the Personal Software Process: Metrics collection and analysis for the differently disciplined. In *Proceedings of the 25th International Conference on Software Engineering*, pages 641–646, 2003.
- [JHC⁺17] Guido Juckeland, Robert Henschel, Sunita Chandrasekaran, Sandra Wienke, Junjie Li, Alexander Bobyr, William Brantley, Mathew Col-

- grove, Oscar Hernandez, Arpith Jacob, Kalyan Kumaran, Dave Raddatz, Veronica Vergara Larrea, Bo Wang, Brian Whitney, Matthias Müller, and Andrey Naraikin. Discussing First Results of the SPEC ACCEL OpenMP Suite with Target Directives. OpenMPCon, 2017.
- [JHJ⁺16] Guido Juckeland, Oscar Hernandez, Arpith C. Jacob, Daniel Neilson, Verónica G. Vergara Larrea, Sandra Wienke, Alexander Bobyr, William C. Brantley, Sunita Chandrasekaran, Mathew Colgrove, Alexander Grund, Robert Henschel, Wayne Joubert, Matthias S. Müller, Dave Raddatz, Pavel Shelepugin, Brian Whitney, Bo Wang, and Kalyan Kumaran. *From Describing to Prescribing Parallelism: Translating the SPEC ACCEL OpenACC Suite to OpenMP Target Directives*, volume 9945, pages 470–488. Springer International Publishing, 2016.
- [JHT16] Andrew Jones, Terry Hewitt, and Owen Thomas. Acquisition and Commissioning of HPC Systems. SC16 Tutorial, 2016.
- [Joh13] P. M. Johnson. Searching under the Streetlight for Useful Software Analytics. *IEEE Software*, 30(4):57–63, 2013.
- [Jon16] Andrew Jones. personal communication, November 13, 2016.
- [JT16] Andrew Jones and Owen Thomas. Essential HPC Finance Practice. SC16 Tutorial, 2016.
- [KBT⁺07] Jonathan Koomey, Kenneth Brill, Pitt Turner, John Stanley, and Bruce Tayloer. A Simple Model for Determining True Total Cost of Ownership for Data Centers. Technical Report TUI3011B, Uptime Institute, 2007.
- [Kep04a] Jeremy Kepner. High performance computing productivity model synthesis. *International Journal of High Performance Computing Applications*, 18(4):505–516, 2004.
- [Kep04b] Jeremy Kepner. HPC productivity: An overarching view. *International Journal of High Performance Computing Applications*, 18(4):393–397, 2004.
- [Kep06a] Jeremy Kepner. High Productivity Computing Systems and the Path Towards Usable Petascale Computing, Part A: User Productivity Challenges. *CTWatch Quarterly*, 2(4A), 2006.
- [Kep06b] Jeremy Kepner. High Productivity Computing Systems and the Path Towards Usable Petascale Computing, Part B: System Productivity Technologies. *CTWatch Quarterly*, 2(4A), 2006.

-
- [KH08] Nurul Naslia Khairuddin and Khairuddin Hashim. Application of Bloom's taxonomy in software engineering assessments. In *Proceedings of the 8th WSEAS International Conference on Applied Computer Science*, pages 66–69, 2008.
- [KKL14] Julian Kunkel, Michael Kuhn, and Thomas Ludwig. Exascale Storage Systems - An Analytical Study of Expenses. *International Journal of Supercomputing Frontiers and Innovations*, 1(1):116–134, 2014.
- [KKS04] Ken Kennedy, Charles Koelbel, and Robert Schreiber. Defining and measuring the productivity of programming languages. *International Journal of High Performance Computing Applications*, 18(4):441–448, 2004.
- [KM09] Kurt Keutzer and Timothy G. Mattson. A Design Pattern Language for Engineering (Parallel) Software. <http://parlab.eecs.berkeley.edu/wiki/patterns/patterns>, 2009.
- [KMM09] John Karidis, Jose E. Moreira, and Jaime Moreno. True Value: Assessing and Optimizing the Cost of Computing at the Data Center Level. In *Proceedings of the 6th ACM conference on Computing frontiers*, pages 185–192, 1531773, 2009. ACM.
- [KMMS10] Kurt Keutzer, Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. A Design Pattern Language for Engineering (Parallel) Software: Merging the PLPP and OPL Projects. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, pages 1–8, 1953620, 2010. ACM.
- [KWA17] Anne Küsters, Sandra Wienke, and Lukas Arnold. *Performance Portability Analysis for Real-Time Simulations of Smoke Propagation Using OpenACC*, pages 477–495. Springer International Publishing, Cham, 2017.
- [Law15] Lawrence Livermore National Laboratory. Hackathon in Computation. <http://computation.llnl.gov/newsroom/cross-pollination-people-and-ideas-summer-hackathon>, 2015.
- [LC08] S. H. Lee and W. Chen. A comparative study of uncertainty propagation methods for black-box-type problems. *Structural and Multidisciplinary Optimization*, 37(3):239–253, 2008.
- [Lea83] Edward E. Leamer. Let's Take the Con Out of Econometrics. *The American Economic Review*, 73(1):31–43, 1983.
- [Lei17] Leibniz Supercomputing Center. SuperMUC Petascale System. <https://www.lrz.de/services/compute/supermuc/systemdescription/>, Accessed January 2017.

- [LGS⁺14] Y. Luo, S. Govindan, B. Sharma, M. Santaniello, J. Meza, A. Kansal, J. Liu, B. Khessib, K. Vaid, and O. Mutlu. Characterizing Application Memory Error Vulnerability to Optimize Datacenter Cost via Heterogeneous-Reliability Memory. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 467–478, 2014.
- [LL17] Dominik Leiner and Stefanie Leiner. SoSci Survey. <https://www.soscisurvey.de>, Accessed in May 2017.
- [LLJ14] J. Legaux, F. Loulergue, and S. Jubertie. Development effort and performance trade-off in high-level parallel programming. In *2014 International Conference on High Performance Computing and Simulation (HPCS)*, pages 162–169, 2014.
- [LMN17] Lars Langner, Maik Maibaum, and Stoyko Notev. LamaPoll. <https://www.lamapoll.com>, Accessed in May 2017.
- [LOVZ97] Am Wasantha Lal, Jayantha Obeysekera, and Randy Van Zee. Sensitivity and uncertainty analysis of a regional simulation model for the natural system in south Florida. In *27th Congress of the IAHR and the ASCE, San Francisco*, pages 560–565, 1997.
- [LRA13] Thomas Ludwig, Albert Reuther, and Amy Apon. Cost-Benefit Quantification for HPC: An Inevitable Challenge. In *Birds of a Feather at the International Conference for High Performance Computing, Networking, Storage and Analysis (SC13)*, 2013.
- [LSG⁺14] Andreas Lintermann, Stephan Schlimpert, Jerry H. Grimmer, Claudia Günther, Matthias Meinke, and Wolfgang Schröder. Massively parallel grid generation on HPC systems. *Computer Methods in Applied Mechanics and Engineering*, 277:131–153, 2014.
- [Lud12] Thomas Ludwig. The Costs of HPC-Based Science in the Exascale Era. In *Invited Talk at the International Conference for High Performance Computing, Networking Storage and Analysis (SC12)*, pages 2120–2188, 2012.
- [LVBS⁺05] Daniel P Loucks, Eelco Van Beek, Jery R Stedinger, Jozef PM Dijkman, and Monique T Villars. *Water resources systems planning and management: an introduction to methods, models and applications*. Paris: Unesco, 2005.
- [McC76] T. J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.
- [McC95] John D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society*

- Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, 1995.
- [McC06] Steve McConnell. *Software Estimation: Demystifying the Black Art*. Microsoft Press, 2006.
- [Mic08] MicroStrategy. Reducing Total Cost of Ownership: Delivering Cost Effective Enterprise Business Intelligence. Technical report, MicroStrategy, 2008.
- [Mil16] Julian Miller. Software Cost Estimation for the Development Effort applied to Multi-node GPU Aeroacoustics Simulation. Master thesis, RWTH Aachen University, 2016.
- [MJ77] John A. Martilla and John C. James. Importance-Performance Analysis. *Journal of Marketing*, 41(1):77–79, 1977.
- [MNV06] Declan Murphy, Thomas Nash, and Lawrence Votta. A System-wide Productivity Figure of Merit. Technical Report SMLI TR-2006-154, Sun Microsystems, Inc., 2006.
- [MNVK06] Declan Murphy, Thomas Nash, Lawrence Votta, and Jeremy Kepner. A System-wide Productivity Figure of Merit. *CTWatch Quarterly*, 2(4B):1–9, 2006.
- [Mod07] J. A. Modi. An introduction to efficiency and productivity analysis, 2nd edition. *Interfaces*, 37(2):198–199, 2007.
- [MRR12] Michael McCool, James Reinders, and Arch Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann Publishers Inc., 2012.
- [MSM04] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, 2004.
- [MSSA12] K. Manakul, P. Siripongwutikorn, S. See, and T. Achalakul. Modeling Dwarfs for Workload Characterization. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, pages 776–781, 2012.
- [MVB08] C. G. Malone, W. Vinson, and C. E. Bash. Data Center TCO Benefits of Reduced System Airflow. *11th IEEE Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems, Vols 1-3*, pages 1199–1202, 2008.
- [MW16] Julian Miller and Sandra Wienke. EffortLog. <http://www.hpc.rwth-aachen.de/research/tco/>, 2016.

- [MWSL⁺17] Julian Miller, Sandra Wienke, Michael Schlottke-Lakemper, Matthias Meinke, and Matthias S. Müller. Applicability of the Software Cost Model COCOMO II to HPC Projects. *International Journal of Computational Science and Engineering*, 2017. Accepted.
- [NB09] Jeffrey Napper and Paolo Bientinesi. Can cloud computing reach the top500? In *Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop*, pages 17–20. ACM, 2009.
- [Nee14] J. Robert Neely. The US Government Role in High Performance Computing: Mission and Policy. Technical Report LLNL-CONF-653229, Lawrence Livermore National Laboratory, 2014.
- [Neu13] Kai Neumann. Comparing the Programmability of Accelerators: OpenMP 4.0 vs. OpenACC. Bachelor thesis, RWTH Aachen University, 2013.
- [New05] MEJ Newman. Power laws, Pareto distributions and Zipf’s law. *Contemporary Physics*, 46(5):323–351, 2005.
- [Nic15] Marco Nicolini. Software Cost Estimation of GPU-accelerated Aeroacoustic Simulations with OpenACC. Bachelor thesis, RWTH Aachen University, 2015.
- [NK03] Edward Nuhfer and Delores Knipp. The Knowledge Survey: A Tool for All Reasons. *To Improve the Academy*, 21:59–78, 2003.
- [NK06] Edward B. Nuhfer and Delores Knipp. Re: The Use of a Knowledge Survey as an Indicator of Student Learning in an Introductory Biology Course. *CBE-Life Sciences Education*, 5(4):313–314, 2006.
- [NMW⁺16] Marco Nicolini, Julian Miller, Sandra Wienke, Michael Schlottke-Lakemper, Matthias Meinke, and Matthias S. Müller. *Software Cost Analysis of GPU-Accelerated Aeroacoustics Simulations in C++ with OpenACC*, pages 524–543. Springer International Publishing, Cham, 2016.
- [Nor10] Geoff Norman. Likert scales, levels of measurement and the “laws” of statistics. *Advances in Health Sciences Education*, 15(5):625–632, 2010.
- [NSB08] Vu Nguyen, Bert Steece, and Barry Boehm. A constrained regression technique for cocomo calibration. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 213–222, 1414040, 2008. ACM.
- [NSNK15] Panagiota Nikolaou, Yiannakis Sazeides, Lorena Ndreu, and Marios Kleanthous. Modeling the Implications of DRAM Failures and

- Protection Techniques on Datacenter TCO. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 572–584. ACM, 2015.
- [Ovi17] Shira Ovide. After Break, Internet Giants Resume Data Center Spending: Gadfly. <http://www.datacenterknowledge.com/archives/2016/05/09/break-internet-giants-resume-data-center-spending-gadfly>, Accessed May 2017.
- [PBF⁺16] Francesca Pianosi, Keith Beven, Jim Freer, Jim W. Hall, Jonathan Rougier, David B. Stephenson, and Thorsten Wagener. Sensitivity analysis of environmental models: A systematic review with practical workflow. *Environmental Modelling & Software*, 79:214–232, 2016.
- [PBG⁺16] Tapasya Patki, Natalie Bates, Girish Ghatikar, Anders Clausen, Sonja Klingert, Ghaleb Abdulla, and Mehdi Sheikhalishahi. *Supercomputing Centers and Electricity Service Providers: A Geographically Distributed Perspective on Demand Management in Europe and the United States*, pages 243–260. Springer International Publishing, Cham, 2016.
- [PCGL07] M. K. Patterson, D. G. Costello, P. F. Grimm, and M. Loeffler. Data center TCO; A comparison of high-density and low-density spaces. Technical report, Intel Corporation, 2007.
- [Pek12] Dmitry Pekurovsky. P3DFFT: a framework for parallel computations of Fourier transforms in three dimensions. *SIAM Journal on Scientific Computing*, 34(4):C192–C209, 2012.
- [PG07a] Rand Perry and Al Gillen. Demonstrating Business Value: Selling to Your C-Level Executives. Technical Report 206363, International Data Corporation (IDC), 2007.
- [PG07b] Rand Perry and Al Gillen. Demonstrating Business Value: Selling to Your C-Level Executives. Technical report, International Data Corporation (IDC), 2007.
- [PG08] I. Patel and J. R. Gilbert. An empirical study of the performance and productivity of two parallel programming models. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–7, 2008.
- [PK04] D. E. Post and R. P. Kendall. Software Project Management and Quality Engineering Practices for Complex, Coupled Multiphysics, Massively Parallel Computational Simulations: Lessons Learned From ASCI. *The International Journal of High Performance Computing Applications*, 18(4):399–416, 2004.

- [Pl010] Dmytro Plotnikov. Produktivität bei der plattformübergreifenden Beschleunigung der Spannungsdickenberechnung für das Kegelradfräsen mit OpenCL. Bachelor thesis, RWTH Aachen University, 2010.
- [Pre00] L. Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, 2000.
- [PS05] Chandrakant D. Patel and Amip J. Shah. Cost Model for Planning, Development and Operation of a Data Center. Technical Report HPL-2005-107, HP Laboratories, 2005.
- [PSV95] Dewayne E Perry, Nancy A Staudenmayer, and Lawrence G Votta. Understanding and Improving Time Usage in Software Development. *Software Process*, 5:111–135, 1995.
- [PSW15] Francesca Pianosi, Fanny Sarrazin, and Thorsten Wagener. A Matlab toolbox for Global Sensitivity Analysis. *Environmental Modelling & Software*, 70:80–85, 2015.
- [QFSH06] Kelly Quinn, Daniel Fleischer, Jed Scaramella, and John Humphreys. Forecasting Total Cost of Ownership for Initial Deployments of Server Blades. Technical Report 202092, International Data Corporation (IDC), 2006.
- [RT06] Albert Reuther and Suzy Tichenor. Making the Business Case for High Performance Computing: A Benefit-Cost Analysis Methodology. *CTWatch Quarterly*, 2(4A):2–8, 2006.
- [Rus17] John Russell. ARM Waving: Attention, Deployments, and Development. HPCwire, <https://www.hpcwire.com/2017/01/18/arm-waving-gathering-attention/>, Accessed May 2017, 2017.
- [Sam01] Rajan Sambandam. Survey of Analysis Methods—Part I: Key Driver Analysis. Quirk’s Marketing Research Review, 2001.
- [SB04] M. Snir and D. A. Bader. A framework for measuring supercomputer productivity. *International Journal of High Performance Computing Applications*, 18(4):417–432, 2004.
- [SBH08] Alan D. Simpson, Mark Bull, and Jon Hill. Identification and Categorisation of Applications and Initial Benchmarks Suite. Technical Report D6.1, Partnership for Advanced Computing in Europe (PRACE), 2008.
- [SBL12] Narendra Sharma, Aman Bajpai, and Mr Ratnesh Litoriya. A comparison of software cost estimation methods: a survey. *The International Journal of Computer Science and Applications (TIJCSA)*, 1(3), 2012.

-
- [SCW⁺13] Dirk Schmidl, Tim Cramer, Sandra Wienke, Christian Terboven, and Matthias S. Müller. Assessing the Performance of OpenMP Programs on the Intel Xeon Phi. In Felix Wolf, Bernd Mohr, and Dieter Mey, editors, *Euro-Par 2013 Parallel Processing*, volume 8097 of *Lecture Notes in Computer Science*, pages 547–558. Springer Berlin Heidelberg, 2013.
- [SD08] Thomas Sterling and Chirag Dekate. Productivity in high-performance computing. *Advances in Computers - High Performance Computing*, 72:101–134, 2008.
- [SGMS16] Lennart Schneiders, Claudia Günther, Matthias Meinke, and Wolfgang Schröder. An efficient conservative cut-cell method for rigid bodies interacting with viscous compressible flows. *Journal of Computational Physics*, 311:62–86, 2016.
- [SIT⁺14] Dirk Schmidl, Christian Iwainsky, Christian Terboven, Christian H Bischof, and Matthias S Müller. *Towards a Performance Engineering Workflow for OpenMP 4.0*, pages 823–832. Parallel Computing: Accelerating Computational Science and Engineering (CSE). IOS Press, 2014.
- [SJSV03] A. Sillitti, A. Janes, G. Succi, and T. Vernazza. Collecting, Integrating and Analyzing Software Metrics and Personal Software Process Data. In *Proceedings of the 29th Euromicro Conference*, pages 336–342, 2003.
- [SKG12] M. Steuwer, P. Kegel, and S. Gorlatch. Towards High-Level Programming of Multi-GPU Systems Using the SkelCL Library. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 1858–1865, 2012.
- [SL17] Michael Schlottke-Lakemper. *A Direct-Hybrid Method for Aeroacoustic Analysis*. Verlag Dr. Hut, 2017. Dissertation at RWTH Aachen University.
- [SLMS15] Michael Schlottke-Lakemper, Matthias Meinke, and Wolfgang Schröder. ZFS. http://www.fz-juelich.de/ias/jsc/EN/Expertise/High-Q-Club/ZFS/_node.html, 2015.
- [SLYB⁺17] Michael Schlottke-Lakemper, Hans Yu, Sven Berger, Matthias Meinke, and Wolfgang Schröder. A fully coupled hybrid computational aeroacoustics method on hierarchical Cartesian meshes. *Computers & Fluids*, 144:137–153, 2017.
- [SMDS15] Erich Strohmaier, Hans W. Meuer, Jack Dongarra, and Horst D. Simon. The TOP500 List and Progress in High-Performance Computing. *Computer*, 48(11):42–49, 2015.

- [Spr12] Paul Springer. A Study of Productivity and Performance of Modern Vector Processors. Bachelor thesis, RWTH Aachen University, 2012.
- [SPW16] Fanny Sarrazin, Francesca Pianosi, and Thorsten Wagener. Global Sensitivity Analysis of environmental models: Convergence and validation. *Environmental Modelling & Software*, 79:135–152, 2016.
- [SRA⁺08] Andrea Saltelli, Marco Ratto, Terry Andres, Francesca Campolongo, Jessica Cariboni, Debora Gatelli, Michaela Saisana, and Stefano Tarantola. *Global Sensitivity Analysis: The Primer*. John Wiley & Sons, 2008.
- [SRK⁺15] Craig A. Stewart, Ralph Roskies, Richard Knepper, Richard L. Moore, Justin Whitt, and Timothy M. Cockerill. XSEDE value added, cost avoidance, and return on investment. In *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, pages 1–8. ACM, 2015.
- [SS10] Caitlin Sadowski and Andrew Shewmaker. The last mile: parallel programming and usability. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 309–314, 1882426, 2010. ACM.
- [Sta17] Standard Performance Evaluation Corporation (SPEC). Standard Performance Evaluation Corporation (SPEC). <https://www.spec.org>, Accessed January 2017.
- [STCR04] Andrea Saltelli, Stefano Tarantola, Francesca Campolongo, and Marco Ratto. *Sensitivity Analysis in Practice: A Guide to Assessing Scientific Models*. Halsted Press, New York, NY, USA, 2004.
- [Ste04] Thomas Sterling. Productivity metrics and models for high performance computing. *International Journal of High Performance Computing Applications*, 18(4):433–440, 2004.
- [Sur14] Andreas Surudo. Application-driven Impact of the Programming Approach on Total Cost of Ownership on the Intel Xeon Phi Architecture—Libraries vs. Intrinsic. Bachelor thesis, RWTH Aachen University, 2014.
- [Swi16] Swiss National Supercomputing Centre (CSCS). GPU Hackathons: EuroHack@Lugano. <https://www.olcf.ornl.gov/training-event/2016-gpu-hackathons/>, 2016.
- [SWM17] Fabian P. Schneider, Sandra Wienke, and Matthias S. Müller. Operational Concepts of GPU Systems in HPC Centers: TCO and Productivity. In *15th International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platform (HeteroPar 2017)*, 2017. Accepted.

-
- [Tan05] Stefan Tangen. Demystifying productivity and performance. *International Journal of Productivity and Performance Management*, 54(1):34–46, 2005.
- [Tas03] Gregory Tasse. Methods for Assessing the Economic Impacts of Government R&D. Technical report, DTIC Document, 2003.
- [TFW⁺16] Abhinav S. Thota, Ben Fulton, Le Mai Weakley Weakley, Robert Henschel, David Y. Hancock, Matt Allen, Jenett Tillotson, Matt Link, and Craig A. Stewart. A PetaFLOPS Supercomputer as a Campus Resource: Innovation, Impact, and Models for Locally-Owned High Performance Computing at Research Colleges and Universities. In *Proceedings of the 2016 ACM on SIGUCCS Annual Conference*, pages 61–68. ACM, 2016.
- [THW13] Jan Treibig, Georg Hager, and Gerhard Wellein. *Performance Patterns and Hardware Metrics on Modern Multicore Processors: Best Practices for Performance Engineering*, pages 451–460. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
- [TKD⁺03] Richard Thomas, Gregor Kennedy, Steve Draper, Rebecca Mancy, Murray Crease, Huw Evans, and Phil Gray. Generic usage monitoring of programming students. In *Proceedings of the 20th Annual Conference of the Australasian Society for Computers in Learning in Tertiary Education (ASCILITE)*, pages 7–10, 2003.
- [TLRW⁺08] Errol Thompson, Andrew Luxton-Reilly, Jacqueline L. Whalley, Minjie Hu, and Phil Robbins. Bloom’s Taxonomy for CS Assessment. In *Proceedings of the tenth conference on Australasian computing education - Volume 78*, pages 155–161. Australian Computer Society, Inc., 2008.
- [TMN⁺99] K. Torii, K. Matsumoto, K. Nakakoji, Y. Takada, S. Takada, and K. Shima. Ginger2: An Environment for Computer-Aided Empirical Software Engineering. *IEEE Transactions on Software Engineering*, 25(4):474–492, 1999.
- [TOP17] TOP500.org. Top500 - The List. <https://www.top500.org>, Accessed January 2017.
- [Tra15] Tiffany Trader. TOP500 Reanalysis Shows ‘Nothing Wrong with Moore’s Law’. HPCwire, <https://www.hpcwire.com/2015/11/20/top500/>, Accessed January 2017, 2015.
- [TS06] W. Pitt Turner and John H. Seader. Dollars per kW plus Dollars per Square Foot Are a Better Data Center Cost Model than Dollars per Square Foot Alone. Technical Report TUI 808, Uptime Institute, 2006.

- [TUD16] TU Dresden and Jülich Supercomputing Center. GPU Hackathons: EuroHack@Dresden. <https://www.olcf.ornl.gov/training-event/2016-gpu-hackathons/>, <https://gcoe-dresden.de/?m=201603>, 2016.
- [VAVS10] A Van Amesfoort, A Varbanescu, and H Sips. Metrics to Characterize Parallel Applications. In *15th Workshop on Compilers for Parallel Computing, Vienna, Austria*, 2010.
- [VFG11] F. Vázquez, J. J. Fernández, and E. M. Garzón. A new approach for sparse matrix vector product on NVIDIA GPUs. *Concurrency and Computation: Practice and Experience*, 23(8):815–826, 2011.
- [VGR09] Kashi Vishwanath, Albert Greenberg, and Daniel A. Reed. Modular Data Centers: How to Design Them? In *Large-Scale System and Application Performance (LSAP)*. Association for Computing Machinery, Inc., 2009.
- [VNL96] Steven P. VanderWiel, Dafna Nathanson, and David J. Lilja. Performance and Program Complexity in Contemporary Network-based Parallel Computing Systems. Technical Report HPPC-96-02, University of Minnesota, 1996.
- [VNL97] Steven P. VanderWiel, Daphna Nathanson, and David J. Lilja. Complexity and Performance in Parallel Programming Languages. In *Proceedings of the Second International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 3–12, 1997.
- [VNL98] Steven P. VanderWiel, Daphna Nathanson, and David J. Lilja. A Comparative Analysis of Parallel Programming Language Complexity and Performance. *Concurrency: Practice and Experience*, 10(10):807–820, 1998.
- [vS11] Thomas von Salzen. Neuer Supercomputer für die RWTH Aachen. idw–Informationsdienst Wissenschaft, <https://idw-online.de/en/news408972>, Accessed May 2017, 2011.
- [WaMM13] Sandra Wienke, Dieter an Mey, and Matthias S. Müller. Accelerators for Technical Computing: Is It Worth the Pain? A TCO Perspective. In Julian Martin Kunkel, Thomas Ludwig, and Hans Werner Meuer, editors, *Supercomputing*, volume 7905 of *Lecture Notes in Computer Science*, pages 330–342. Springer Berlin Heidelberg, 2013.
- [WCM16] Sandra Wienke, Tim Cramer, and Matthias S. Müller. Software Lab: Parallel Programming Models for Applications in the Area of

- High-Performance Computation. Chair for High Performance Computing, RWTH Aachen University, <http://www.hpc.rwth-aachen.de/teaching>, 2013-2016.
- [WCMS15] Sandra Wienke, Tim Cramer, Matthias S. Müller, and Martin Schulz. Quantifying Productivity—Towards Development Effort Estimation in HPC. Poster at the International Conference for High Performance Computing, Networking, Storage and Analysis (SC15), 2015.
- [WiaMM15] Sandra Wienke, Hristo Iliev, Dieter an Mey, and Matthias S. Müller. Modeling the Productivity of HPC Systems on a Computing Center Scale. In Julian M. Kunkel and Thomas Ludwig, editors, *High Performance Computing*, volume 9137 of *Lecture Notes in Computer Science*, pages 358–375. Springer International Publishing, 2015.
- [Wie16a] Sandra Wienke. Development Effort & Productivity Estimation in HPC. Chair for High Performance Computing, RWTH Aachen University, <http://www.hpc.rwth-aachen.de/research/tco>, 2016.
- [Wie16b] Sandra Wienke. Development Effort Methodologies. Chair for High Performance Computing, RWTH Aachen University, <http://www.hpc.rwth-aachen.de/research/tco>, 2016.
- [Wil16] G Wilson. Software Carpentry: lessons learned [version 2; referees: 3 approved]. *F1000Research*, 3(62), 2016.
- [Wis12] Wissenschaftsrat. Empfehlungen zur Förderung von Forschungsbauten (2013). Technical Report Drs. 2222-12, 2012.
- [Wis14] Wissenschaftsrat. Empfehlungen zur Förderung von Forschungsbauten (2015). Technical Report Drs. 3781-14, 2014.
- [Wis15a] Wissenschaftsrat. Empfehlungen zur Finanzierung des Nationalen Hoch- und Höchstleistungsrechnens in Deutschland. Technical Report Drs. 4488-15, 2015.
- [Wis15b] Wissenschaftsrat. Empfehlungen zur Förderung von Forschungsbauten (2016). Technical Report Drs. 4548-15, 2015.
- [WK11] Lizhe Wang and SameeU. Khan. Review of performance metrics for green data centers: a taxonomy study. *The Journal of Supercomputing*, pages 1–18, 2011.
- [WMSM16] Sandra Wienke, Julian Miller, Martin Schulz, and Matthias S. Müller. Development Effort Estimation in HPC. In *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 107–118, 2016.

- [WP05] Karl R. Wirth and Dexter Perkins. Knowledge Surveys: An Indispensable Course Design and Assessment Tool. *Innovations in the Scholarship of Teaching and Learning*, 2005.
- [WPM⁺11] Sandra Wienke, Dmytro Plotnikov, Dieter Mey, Christian Bischof, Ario Hardjosuwito, Christof Gorgels, and Christian Brecher. Simulation of bevel gear cutting with GPGPUs—performance and productivity. *Computer Science - Research and Development*, 26(3-4):165–174, 2011.
- [WSD⁺14] Sandra Wienke, Marcel Spekowius, Alesja Dammer, Dieter an Mey, Christian Hopmann, and Matthias S. Müller. Towards an accurate simulation of the crystallisation process in injection moulded plastic components by hybrid parallelisation. *International Journal of High Performance Computing Applications*, 28(3):356–367, 2014.
- [WSTaM12] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. OpenACC - First Experiences with Real-World Applications. In *Euro-Par 2012 Parallel Processing*, volume 7484 of *Lecture Notes in Computer Science*, pages 859–870. Springer Berlin Heidelberg, 2012.
- [WTaMM13] Sandra Wienke, Christian Terboven, Dieter an Mey, and Matthias S. Müller. Accelerators, Quo Vadis? Performance vs. Productivity. In Waleed W. Smari and Vesna Zeljkovic, editors, *Proceedings of the 2013 International Conference on High Performance Computing & Simulation (HPCS 2013)*, pages 471–473. IEEE, 2013.
- [WTBM14] Sandra Wienke, Christian Terboven, James C. Beyer, and Matthias S. Müller. A Pattern-Based Comparison of OpenACC and OpenMP for Accelerator Computing. In *Euro-Par 2014 Parallel Processing*, volume 8632 of *Lecture Notes in Computer Science*, pages 812–823. Springer International Publishing, 2014.
- [YZ10] Sheng Yu and Shijie Zhou. A survey on metric of software complexity. In *2010 2nd IEEE International Conference on Information Management and Engineering*, pages 352–356, 2010.
- [ZH09] M. Zhang and L. Hochstein. Fitting a workflow model to captured development data. In *3rd International Symposium on Empirical Software Engineering and Measurement*, pages 179–190, 2009.
- [ZKI13] Zentrum für Kommunikation und Informationsverarbeitung e.V. - Arbeitskreis Supercomputing. Weiterentwicklung des Hochleistungsrechnens in Deutschland (Positionspapier). http://www.zki.de/fileadmin/zki/Publikationen/HPC-Positionspapier_ZKI.pdf, 2013.

Acronyms

AAT	all-at-a-time
AD	automatic differentiation
ANOVA	analysis of variance
BCR	benefit-cost ratio
BI	business intelligence
BLAS	Basic Linear Algebra Subprograms
Capex	capital expenditures
CBA	cost-benefit analysis
ccNUMA	cache-coherent NUMA
CDF	cumulative distribution function
CE	cost effectiveness
CEA	cost-effectiveness analysis
CG	Conjugate Gradient
CLAIX	Cluster Aix-la-Chapelle
COCOMO	Constructive Cost Model
CPE	Computing Processing Element
CPU	central processing unit
CRS	compressed row storage
CSV	comma-separated values
CUDA	Compute Unified Device Architecture
DARPA	Defense Advanced Research Projects Agency
DCF	discounted cash flow
DFG	German Science Foundation
DG	Discontinuous Galerkin
DIMM	dual in-line memory module
DKRZ	German Climate Computing Center

DNS	Direct Numerical Simulation
DOE	Department of Energy
DRAM	dynamic random-access memory
ECC	error-correcting code
EI	external inputs
EIF	external interface files
EM	effort multiplier
EO	external outputs
EQ	external inquiries
FFI	factor fixing
FFT	Fast Fourier Transform
FP	function point
FPR	factor prioritization
FTE	full-time equivalent
GCS	Gauss Centre for Supercomputing
GDP	gross domestic product
GPU	graphics processing unit
GSA	global sensitivity analysis
GSC	general system characteristic
GUM	guide to the expression of uncertainty in measurement
HDD	hard disk drive
HPC	High-Performance Computing
HPCC	HPC Challenge
HPCG	HPC Conjugate Gradients
HPCS	High Productivity Computing Systems
HPL	High Performance Linpack
IDC	International Data Corporation
IFPUG	International Function Point Users Group
ILF	internal logical files
I/O	input/output
IRR	internal rate of return

ISA	instruction set architecture
IT	information technology
JARA	Jülich Aachen Research Alliance
JSON	JavaScript Object Notation
KS	knowledge survey
LHS	Latin Hypercube sampling
LLNL	Lawrence Livermore National Laboratory
LLOC	logical lines of code
LOC	lines of code
MFP	multifactor productivity
MMRE	mean magnitude of relative error
MPE	Management Processing Element
MPI	Message Passing Interface
MTTF	mean time to failure
NASA	National Aeronautics and Space Administration
NHR	Nationales Hoch- und Höchstleistungsrechnen
NPB	NAS Parallel Benchmark
NPV	net present value
NSF	National Science Foundation
NUMA	non-uniform memory access
OAT	one-at-a-time
OpenMP	Open Multi-Processing
Opex	operational expenses
OS	operating system
PBS	productivity benchmark suite
PDF	probability density function
PLC	performance life-cycle
PRACE	Partnership for Advanced Computing in Europe
PUE	power usage effectiveness
PV	present value
RAID	redundant array of independent disks

RAPL	Running Average Power Limit
RAS	reliability, availability and serviceability
RDTP	relative development time productivity
ROI	return on investment
RWTH	Rheinisch-Westfälische Technische Hochschule
SA	sensitivity analysis
SE	software engineering
SEI	Software Engineering Institute
SF	scale factor
SLOC	source lines of code
SPEC	Standard Performance Evaluation Corporation
SpMV	sparse matrix-vector multiplications
SSD	solid-state disk
TCO	total costs of ownership
TDP	thermal design power
UFC	unadjusted function points count
UPC	Unified Parallel C
UPS	uninterruptible power supply
VAF	value adjustment factor
VBA	variance-based approach
VBSA	variance-based sensitivity analysis
XSEDE	Extreme Science and Engineering Discovery Environment
ZKI	Association of German University Computing Centers

Appendix

A.1. Use Cases

In the context of my research, various case studies have been used to investigate productivity, effort, performance or TCO. For this thesis, I pick three real-world applications to illustrate the applicability of my methodologies: *NINA*, *psOpen* and *ZFS*. They represent different classes of applications: (a) small codes running on a single compute node to evaluate different setups like architectures or programming models, (b) huge large-scaling codes that have been targeted for performance tuning on a known architecture, and (c) huge large-scaling codes that have been ported to a new (accelerator-type) architecture and that use object-oriented code design. In addition, a *Conjugate Gradient Solver* serves as compact use case for classroom experiments.

A.1.1. Neuromagnetic Inverse Problem — NINA

This software for the solution of Neuromagnetic *IN*verse *LA*rge-scale problems (*NINA*) represents a class of applications that runs on a single compute node. It contains three (similarly-structured) kernels that account for roughly 90 % of the serial execution time. Its kernels benefit from parallelization on x86- and accelerator-type architectures. Since the kernels amount to only ~ 100 LOC overall, this application is well-suited to investigate these different architecture setups.

Experiences with the parallelization and tuning of this bio-medical application on different hardware architectures have been subject in several of my publications [WSTaM12][WaMM13][WTaMM13][SCW⁺13][SWM17]. In the following, an overview of the application’s setup, the needed effort and the gained performance is given. It is extended with data gathered from further parallelizations by developers, student workers and apprentices who tackled recent hardware architectures or novel parallel programming models.

Description

The application *NINA* comes from the field of bio-medicine or more precisely magnetoencephalography and aims at reconstructing the focal activity of a hu-

man brain. For that, test persons equipped with electrodes (on their heads) get different stimuli. The corresponding current density inside the human brain can be measured outside the head by an induced magnetic field. The resulting neuromagnetic inverse problem can be solved by means of a minimum p -norm solution. To tackle the challenge of computational efficiency and accuracy effecting the convergence behavior of this unconstrained nonlinear optimization problem, the software package of Bücker et al. [BBR08] employs first- and second-order derivatives with automatic differentiation (AD).

While, the software package is implemented primarily in Matlab, the problem's objective function, its first- and second-order derivatives are written in C to enable AD combined with parallel computing. These three functions represent the kernels of the application. The Matlab program calls the kernels about thousand times during the optimization process. For simplification, a C framework was established that mimics the original call hierarchy. The three kernels include mainly computations of matrix-vector products using a matrix of dimensions $128 \times 512,000$. The special structure of the matrix arising from the medical setup cannot be handled well by available BLAS libraries so that manual parallelization and tuning is needed. Additionally, the matrix can be divided into a big dense and a small sparse part.

HPC Activities

The *NINA* application has been parallelized using OpenMP for recent x86-based processors, as well as OpenACC and CUDA for NVIDIA GPUs.

The first parallel version of the code could be straightforward implemented with OpenMP by parallelizing all matrix-vector and vector-vector operations. Performance tuning techniques applied to this version cover blocking of matrix-vector multiplications, vectorization, data alignment on page size and ensuring data affinity on NUMA domains. The OpenACC version similarly parallelizes all vector operations, but needed slight restructuring of data accesses to prevent data races. In addition, kernels are executed asynchronously to the host by using 16 streams. Data transfer times have been optimized by using pinned memory. The CUDA version has been implemented analogously to the OpenACC version and extended by highly-optimized reduction operations. Furthermore, dynamic parallelism has been leveraged and the asynchronous execution has been further tuned to minimize interaction with the host.

Application-Dependent Productivity Data

With respect to the HPC activities given above, the development efforts are based on a moderately-experienced programmer who already knows details on the hard-

Table A.1.: Application-dependent productivity parameters of *NINA*: effort, performance and power consumption for different HPC setups. System details are given in Tab. A.4.

System	Programming model	Effort [days]	Kernel runtime [s]	Av. power [W]: serial portion	Av. power [W]: parallel portion
SNB	OpenMP	5	44.71	19	448
K20	OpenACC	4	29.72	211	337
K20	CUDA	6	22.37	211	335

ware architectures and the programming paradigms. Since some code lessons learned during the first implementation could directly be applied to other code versions, some approximated time is added to the real-measured one to be able to compute costs independently. The development efforts that were needed to create the various code versions can be found in Tab. A.1: It took 5 days to implement the highly-tuned OpenMP version. The development time with OpenACC was even a bit shorter than with OpenMP because the PGI OpenACC compiler generated a GPU version with reasonable performance with less code restructuring. The CUDA version took more effort due to its low-level character. Since the *NINA* kernels are very small, any annual maintenance effort is ignored.

The runtime of the corresponding *NINA* applications is measured 100 times and from that the average runtimes are derived. Here, it is distinguished between runtime of the (parallelized) kernel code and runtime of the serial portion of the code which includes reading input data, writing output data and triggering the accelerator (if applicable). While the runtime of the serial code portion remains the same for all implementations, the kernel runtimes differ and are summarized in Tab. A.1. It is noteworthy that for this single-node setup, the node capacity is $n_i = 1$ and that, hence, a constant quality weighting factor $q = 1$ is assumed.

For setting up the application’s energy model, the differentiation between serial and kernel code portion is used as described in Equ. (7.3). For each, the average power consumption in watt is measured and multiplied by its respective runtime share. The power measurements are carried out using the LMG450 power meter that traces the power consumption of the application over time.

A.1.2. Engineering Application — psOpen

The code *psOpen* simulates fine-scale turbulences and belongs to the class of large-scaling applications [GG15]: As part of the High-Q Club, it scales up to $\sim 460,000$ cores on a BlueGene/Q system. It consists of roughly 15,000 to 20,000 lines of code

from which the kernel under investigation amounts to approximately 1,000 LOC. This main kernel takes up to 80 % of the total execution time and contains the application's network communication.

The performance of this application has been tuned in the context of activities of the cross-sectional JARA-HPC group [JAR17c] for an x86 architecture. Based on this, the corresponding productivity evaluations are covered in [WIaMM15]. Here, a short summary of this work is given.

Description

The engineering application *psOpen* is a hybrid MPI+OpenMP code written in C/Fortran and simulates turbulent flows in the field of combustion technology. It implements a pseudo-spectral Direct Numerical Simulation (DNS) method that solves the discretized Navier-Stokes equations for incompressible fluids in reciprocal (Fourier) space. Since one part of the solver still works in real space, a three-dimensional Fast Fourier Transform (3D-FFT) switches between the two representations. This 3D-FFT represents the most time-consuming kernel of the application. It communicates over network, while the rest of the algorithm performs local point operations only.

HPC Activities

The HPC activities examined in this work are based on the parallel MPI+OpenMP implementation of *psOpen* and, thus, cover performance tuning activities rather than (initial) parallelization steps. These optimization activities have been performed in two phases. In the first phase, modifications to the open-source library *P3DFFT* [Pek12] that is used by *psOpen* has been applied by the application developer: The filtering step of the pseudo-spectral DNS method with the Fourier transform has been fused and, thus, the data amount sent across the network has been significantly reduced with respect to the original full 3D-FFT. In the second phase, an HPC expert carried out an analysis and performance model creation of the communication pattern of the 3D-FFT routine. From the performance model, an optimized domain decomposition could be extracted.

Application-Dependent Productivity Data

Each of the optimization phases required effort of one person-month, meaning 17.5 person-days. Effort numbers are based on the work of experts in the application domain and high-performance computing as mentioned above. A summary of the corresponding values can be found in Tab. A.2.

Table A.2.: Application-dependent productivity parameters of *psOpen*: effort, performance and power consumption for two implementation phases.

Phase	Effort [days]	Kernel speedup	Av. power [W]
1	17.5	1.69	275
2	17.5	1.82	275

To estimate the runtime behavior across a large number of compute nodes, a performance model was set up by the engaging HPC expert [WIaMM15]: $t(n) = t_{\text{io}} + t_{\text{pc}}(n) + t_{\text{FFT}}(n)$. Here, $t_{\text{io}} = c_{\text{io}} \cdot t_1$ represents the serial I/O runtime. The parallel time for local point computations is modeled by $t_{\text{pc}}(n) = c_{\text{pc}} \cdot t_1/n$ and the runtime of the main FFT kernel is given by $t_{\text{FFT}}(n) = c_{\text{FFT}} \cdot t_1/(\text{ksp} \cdot n)$ where ksp represents the kernel speedup (see Tab. A.2), t_1 the overall serial runtime and n the number of nodes. The respective runtime fractions are approximated by $c_{\text{io}} = 0.02\%$, $c_{\text{pc}} = 19.98\%$, and $c_{\text{FFT}} = 80.00\%$ for a domain size of 4096^3 points. Due to the large-scale character of this application, $n_{\text{scale}} = n_i$ is set to 1,000. For simplicity, a quality weighting factor of 1 is assumed while more complex factors have been tested in [WIaMM15].

For the application-dependent energy model, the average application’s power consumption per node is taken multiplied by the corresponding runtime. This simplification is made due to the high parallelism within the application that yields almost a constant utilization.

A.1.3. Conjugate Gradient Solver

The Conjugate Gradient (CG) code under investigation belongs to the class of applications running on a single compute node. It solves a system of linear equations for symmetric positive-definite matrices. The original (serial) CG solver code in this work is written in C and contains roughly 1,000 lines of code, from which ~ 150 lines belong to the actual solver. Sparse matrix-vector multiplications (SpMV) represent 73% of the solver time. Due to its small size and well-known algorithm, the CG solver is well-suited to be parallelized on tuned on different architectures.

The CG method has been covered in many publications and is applied in numerous real-world applications. The serial C version of the code was created by Cramer who presents a parallel extension of this code in [CSKaM12]. The CG code has been subject of student software labs at RWTH Aachen University under supervision of Cramer and me [WCM16]. In [MWSL⁺17], this software lab data has been analyzed with respect to efforts spent by students. Furthermore, a hybrid multi-GPU-CPU version of the code has been used to evaluate the productivity of

various operational concepts of GPU systems [SWM17]. Since the CG method is well known, it is not explicitly introduced here and only an overview of its data set is given. Furthermore, its application in student software labs is described in more detail, and HPC activities and performance of reference versions are presented.

Description

The CG code examined in this work follows an iterative implementation with residual $< 10^{-8}$ as convergence criterion. The sparse matrix data set *AMD/G3_circuit* is taken from the University of Florida Sparse Matrix Collection [DH11] and comes from a circuit simulation problem. It contains 7,660,826 non-zeros and is either represented in CRS format or in ELLPACK-R format [VFG11]. Respectively, the matrix has a memory footprint of 135 MB or 165 MB.

Application in Student Software Labs In software labs at RWTH Aachen University [WCM16], students are supposed to manually parallelize the serial C version of the code with three parallel programming models: OpenMP on two-socket Intel Westmere processors (see WST in Tab. A.4), OpenACC and CUDA on NVIDIA Fermi GPUs (FQU in Tab. A.4). Before attending the class, students got an introduction to parallel programming with the three programming models so that they were able to solve the given assignments. Students mostly worked in teams of two and were (partly) graded based on (the performance of) their corresponding implementations. In 2013 and 2014, the serial base code set CRS as matrix data format in advance with its naive matrix access pattern being more suitable for CPUs than for GPUs. However, students were allowed to change this data format for performance reasons. In addition, students could choose the order of implementation with respect to OpenMP, OpenACC and CUDA, where most participants started with OpenMP. In 2015 and 2016, the matrix storage format was preset to ELLPACK-R where the naive matrix access pattern favors GPUs over CPUs. Again, students had the freedom to modify this storage format for certain implementations. In these years, the order of applying the three parallel programming models was rather predefined since intermediate tests required a certain order. The ordering options represented a complete permutation across the three programming models. Over all four years, 25 student teams handed in solutions with numerically correct results.

As another part of the classroom experiment, students in the software labs were asked to track their development efforts for each parallel programming model including time spent for familiarization with the code and hardware. In 2013 and 2014, students kept manual developer diaries for that with information on development time, the number of modified lines of code, achieved performance in

terms of the CG solver's runtime, and free-form comments. In 2015 and 2016, students had to use the *EffortLog* tool (compare Sect. 11.2) to track effort, milestones and performance gains. To evaluate the students' knowledge before the class and their corresponding learning effects, students were asked to voluntarily participate at pre- and post-knowledge surveys as introduced in Sect. 10.4. By filling out pre-knowledge surveys, they could get some kind of preview on intermediate test questions. Across all students that also took part at the post-knowledge survey, a book prize was raffled as motivation.

HPC Activities

Students in software labs provided numerous different solutions for the parallelization of the CG method. Additionally, as comparison for performance results, a highly-tuned reference version for each programming model has been implemented and steadily improved over four years. The corresponding HPC activities are presented here.

The OpenMP reference version uses the CRS storage format and optimizes load balancing by a task-driven approach. Data affinity has been tuned for NUMA-featuring compute nodes by applying first touch principles and thread binding. The usage of a Jacobi preconditioner reduces the number of iterations and, thus, improves runtime. It has been applied to all code versions. To leverage GPU computational power with OpenACC, all loops within the CG solver have been off-loaded as kernels. Corresponding kernel launch configurations and loop schedules have been optimized. Data transfers between CPU and GPU have been minimized, and optimized using pinned memory. Memory transactions within the GPU make use of (texture) caches and access matrix data based on the ELLPACK-R storage format. Furthermore, computations and data transfers have been overlapped by applying the (asynchronous) streaming concept. For CPU-GPU hybrid computations, OpenACC with ELLPACK-R remains the approach on the GPU, while OpenMP with CRS data is used for parallel execution on the CPU. The matrix and vectors are row-wise decomposed in two chunks among the devices where the chunk sizes are experimentally determined for best performance. The CUDA reference version applies the same concepts as the OpenACC (hybrid) implementation.

Application-Dependent Productivity Data

Students provided their effort logs with manual or electronic developer diaries. Since the reported development times differ greatly across students, the efforts needed to implement the aforementioned reference versions are presented here and base on an experienced programmer (compare Tab. A.3): The OpenMP im-

Table A.3.: Application-dependent productivity parameters of the *CG solver*: reference effort and performance for (hybrid) implementations with different parallel programming models.

Programming model	Effort [days]	Runtime of solver [s]	GFlop/s of SpMV
OpenMP	1	7.94	3.48
OpenACC	2	3.62	8.43
OpenACC+OpenMP	3	3.39	10.68
CUDA	3	3.18	9.31
CUDA+OpenMP	5	2.97	10.21

Table A.4.: Hardware details for different architectures used throughout this work.

System	Hardware details
WST	2-socket Intel Westmere X5675 @3.07 GHz, 2x 6 cores, 24 GB memory
SNB	2-socket Intel Sandy Bridge E5-2680 @2.7 GHz, 2x 8 cores, 64 GB memory
BDW	2-socket Intel Broadwell E5-2650 v4 @2.2 GHz, 2x 12 cores, 128 GB memory
FQU	NVIDIA Fermi Quadro 6000 GPU with 2-socket Intel Westmere X5650 @2.67 GHz host, 448 cores, 6 GB memory, ECC on
K20	NVIDIA Kepler K20Xm GPU with SNB host, 896 cores, 6 GB memory, ECC on
P100	NVIDIA Pascal P100 SXM2 with BDW host, 3584 cores, 16 GB memory, ECC on

plementation took roughly 1 person-day, while the both GPU versions needed 2 to 3 days. Creating hybrid versions of the codes yielded additional effort.

The performance results in Tab. A.3 have been obtained using the Intel 16.0 compiler for OpenMP, PGI 16.9 compiler for OpenACC and the CUDA toolkit 7.5 for CUDA. All measurements represent the minimum of ten repetitions. The GPU versions achieved higher (absolute) performance than the OpenMP version with slight additional performance improvements in hybrid execution.

A.1.4. Hardware Setup

For all case studies under investigation, hardware from the RWTH Compute Cluster was used. This includes standard *MPI nodes* of the RWTH Bull Cluster installed in 2011 [vS11], i.e., 2-socket Intel Westmere servers, referred to as WST

in Tab. A.4. In addition, hardware powering the RWTH’s *aixCave* [ITC17b] installed in 2012 was exploited for GPU-based codes and comprises two NVIDIA Fermi Quadro GPUs per node (see FQU in Tab. A.4). Until 2017, these nodes were employed in a dual-purpose configuration: on day time, they support virtual reality activities in the *aixCave*, and on night time, they served as compute nodes for batch jobs. Further RWTH compute nodes used contain NVIDIA Kepler GPUs and 2-socket Intel Sandy Bridge CPUs (K20 and SNB in Tab. A.4). The RWTH’s *CLAIx* cluster part (installed in 2016) provides Intel Broadwell standard servers (compare BDW in Tab. A.4) and NVIDIA Pascal GPUs (P100).

A.2. Productivity Evaluation

A.2.1. Parameters & Predictability

Table A.5.: Parameters (P) of productivity model categorized (C) by application-dependent (A) and system-dependent (S) components. Estimations can usually be conducted by managers (M), managers with input from vendors (V) or application developers (U).

P	Definition	C	Prediction possibility
n	number of compute nodes	-	often fixed investment given (M), hardware prices (V)
τ	system lifetime	-	contractual maintenance periods (V), breakdown of electrical components
α	system availability	S	previously-experienced system availabilities on that site (M), MTTF, reliability, availability, serviceability (V)
m	number of relevant applications	S	tendering joint applicants, tendering job mix, previously-experienced core-h statistics (M)
p	capacity-based weighting factor	A	experiences from previous cluster usage (accounting information) (M)
t_{app}	runtime of application	A	performance model (U), hardware characteristics (V)
q_{app}	quality weighting factor	A	interview with user (U)
sl	salary of FTE	-	given by the employer
$C_{HW}^{ot,n}$	HW acquisition	S	interviews (V)
$C_{HM}^{pa,n}$	HW maintenance	S	determined by p_{HM}

Table A.5.: *Continued from previous page*

P	Definition	C	Prediction possibility
p_{HM}	HW maintenance percentage	S	interviews (V)
$C_{IF}^{ot,n}, C_{IF}^{pa,n}$	building/ infrastructure	S	interviews (V), determination of limiting factor for housing machinery: express in this unit (M)
$C_{VE}^{ot,n}, C_{EV,nt}^{ot}$	OS/ env. installation	S	mostly determined by AE and sl
AE	administration effort	S	previous experiences, interviews with administrators (M)
$C_{VM}^{pa,n}, C_{EM}^{pa,nt}$	OS/ env. maintenance	S	previous experiences, interviews with administrators (M)
$C_{SW}^{pa,nt}$	compiler/ software	S	interview with compiler/ software vendors or resellers (V)
$C_{DE}^{ot,nt}$	HPC development	A	determined by DE and sl
DE	development effort	A	HPC software cost model (U)
$C_{DM}^{pa,nt}$	HPC application maintenance	A	determined by DM and sl
DM	application maintenance effort	A	software cost model (U)
$C_{EG}^{pa,n}$	energy	A	determined by co , ec , pue
co	power consumption	A	power model (U), interviews (V)
ec	electricity	S	contracts with electricity provider (M)
pue	power usage effectiveness	S	previous experiences, interviews (V), infrastructure restrictions (M)

A.2.2. Parameters for RWTH Aachen University Setups

Productivity parameters experienced at the IT Center of RWTH Aachen University have been quantified and described in Wienke et al. [WaMM13] and Bischof et al. [BaMI12]. I summarize these basic parameter assumptions that I use for case studies in this work. An overview of all parameters can be found in Tab. A.6.

Looking on the one-time costs and especially the hardware purchase costs, we use list prices provided by the company Bull in January 2013. To estimate the infrastructure costs for housing the compute devices, a depreciation of the actual

one-time building costs of 5 M€ (without offices) is assumed over 25 years [BaMI12] and, thus, yields annual cost of 200,000 €. Breaking it down on a per node basis, we divide it by 1.6 MW of electrical supply which is the limiting factor for housing machinery in the building [BaMI12], and multiply it by the maximum power consumption of a compute node. The initial environment costs include administration effort for integrating and installing systems, as well as activities to ensure that further maintenance updates can be easily rolled out to all nodes of a system type simultaneously. Both, administration and programming efforts are transferred to manpower costs by multiplying it with the cost of one day of an FTE. We define the cost of one person-day to be 272.86 € in accordance to funding guidelines of the DFG in 2013 [Ger13] that suggest an annual salary for doctoral researchers (or comparable) of 57,300 € and further assume 210 working days per year with respect to the European Commission’s CORDIS [Eur14].

For the annual costs, we account 2 % of the netto purchasing costs as hardware maintenance that is provided by the vendor. The maintenance of the operating system and compute environment of the whole RWTH compute cluster is employed by four administrative FTEs [BaMI12] in 75 % of their time. We break down the administration effort per node by dividing the 180,000 € manpower costs by the total number of nodes in our cluster (roughly 2,300) and get approximately 78 € per any kind of compute node. There is no significant additional effort per node type since a generic approach to roll out software was established during the first installation. Software and compiler costs account roughly to 50,000 € per year at the IT Center including compiler and tool expenses. However, when assuming an integrative hosting concept [ITC16] for the single-application perspective, RWTH institutes do not have additional compiler or tools costs. Therefore, most configurations under analysis assume annual software costs of 0 €. Energy costs are dependent on the hardware, the running application, the PUE of the computing center and the regional electricity cost. The IT Center pays roughly 0.15 €/kWh and estimates its PUE to 1.5 [BaMI12] in 2013. The power consumption was measured either with a Raritan Dominion PX power distribution unit or with an LMG450 power meter. For simplicity, we take the application’s power consumption with respective distribution to its serial and parallel runtime share for the whole system lifetime including system unavailabilities (compare Equ. (7.4)). The costs for application maintenance are again determined by the annual maintenance effort for the application multiplied by the FTE salary. Here, we assume for most setups the application maintenance costs of 0 €.

Table A.6.: Summary of estimated or measured productivity parameters of *psOpen* and *NINA*. Parameters that are varied in SA are marked in column V(ariet). Color coding is just for better readability.

	V	Parameter	Small scale	Large scale	HW comparison			
	×	n	56	500	$I = 250,000 \text{ €}$			
	×	τ [years]	5					
application-dependent		application prog. model	$psOpen$ MPI+OpenMP		$NINA$ OpenMP OpenACC CUDA			
	×	$n_i = n_{\text{scale}}$ $t_{app}(n)$ [s]	1000 $0.0002 \cdot t_1 +$ $0.1998 \cdot t_1/n +$ $0.8 \cdot t_1/(\text{ksp} \cdot n)$		1 45.15 + 45.15 + 45.15 + 44.71 29.72 22.37			
	×	t_1 [s]	1000		-			
	×	ksp	1.82		-			
	×	DE [days]	35	210	5	4	6	
	×	CO_{ser} [W]	275		193.44	211.44		
	×	CO_{par} [W]	275		447.99	336.67	334.92	
	×	DM [days]	0	10	0			
		q_{app}	1		1			
	system-dependent		system	WST		SNB	K20	
×		α			80 %			
×		sl [€/day]			272.86			
×		ec [€/kWh]			0.15			
×		pue			1.5			
×		$C_{HW}^{ot,n}$ [€]			no public information			
		$C_{IF}^{ot,n}$ [€]			0			
×		$C_{VE}^{ot,n}$ [€]			0			
×		$C_{EV}^{ot,nt}$ [€]			0			
×		p_{HM}	2 %		8.2 %			
		$C_{HM}^{pa,n}$ [€]			$C_{HW}^{ot,n}/1.19 \cdot p_{HM}$			
×		$C_{IF}^{pa,n}$ [€]			$C_{building}^{pa} \cdot CO_{max}/CS_{supply}$			
		$C_{building}^{pa}$ [€]			200,000			
		CO_{max} [W]	290		629	795		
		CS_{supply} [W]			1,600,000			
×		$C_{VM}^{pa,n}$ [€]			180,000 € / 2,300 = 78			
×		$C_{EM}^{pa,nt}$ [€]			0			
		$C_{DE}^{ot,nt}$ [€]			$sl \cdot DE$			
		$C_{DM}^{pa,nt}$ [€]			$sl \cdot DM$			
×		$C_{SW}^{pa,nt}$ [€]			0			

Input

Fixed parameters for Total Cost of Ownership

⚠ Please include taxes in all costs. ⚠

	one-time	annual
Hardware	4464.28 €	89.29 €
per node Building / infrastructure	0.00 €	36.25 €
OS / environment	0.00 €	78.00 €
per node type - OS / environment	0.00 €	0.00 €

Additional annual costs

Energy per node 542.00 € / year

Compiler / software 0.00 €

Programming

Person day 272.86 €

Application maintenance 0 person days

System availability 80 %

Parameters for the program

Simulation

Export / Import

Input

Fixed parameters for Total Cost of Ownership

Parameters for the program

Initial tuning

Effort 17.5 person days

Lines of Code 1000

Kernel speedup 1.69

Performance 80.5 %

Formulas

s in COCOMO II 1.16

k in 80-20-rule 0.005

Runtime

☐ Amdahl's law

Serial runtime 1000 s

Parallel part 80 %

⚙ Custom formula

The formula is evaluated in dependence of n that means the number of nodes and the kernel speedup k_{sp} . In addition you can use math functions from JavaScript, for example `Math.floor()`.

$0.2 + 800 / k_{sp} / n + 199.8 / n$

⚠ This code will be executed in your browser. Be sure to only paste trusted content! ⚠

Simulation

Export / Import

Input

Fixed parameters for Total Cost of Ownership

Parameters for the program

Simulation

⚠ Please include taxes in all costs. ⚠

☒ System lifetime and effort

Fixed investment 750000.00 €

Lifetime (maximum) 150 years

Effort (maximum) 150 person days

☐ Number of nodes and effort

System lifetime 5 years

Number of nodes (maximum) 150 nodes

Effort (maximum) 150 person days

Output

☒ Runtime ☐ Productivity

☐ Maximum investment 750000.00 €

☐ Effort

Number of nodes 56 nodes

System lifetime 5 years

Effort (maximum) 50 person days

Output

☒ Productivity ☐ Performance ☐ Kernel speedup

☐ Nodes

Effort 17.5 person days

System lifetime 5 years

Number of nodes (maximum) 56 nodes

☐ Second phase in multi-phase cluster

Investment for first phase 5000000.00 €

Investment for second phase 5000000.00 €

Targeted system lifetime 8 years

Start of second phase 7 years

Simulation ☒ Start of second phase ☐ Productivity with fixed start of second phase

Runtime formula for the second phase

The formula is evaluated in dependence of n that means the number of nodes and the start of the second phase δ .

In addition you can use math functions from JavaScript, for example `Math.pow()`.

⚠ The runtime for the first phase is evaluated without tuning effort. You may want to adapt the formula in "Parameters for the program". ⚠

$0.2 + 800 / 1.69 / n + \text{Math.pow}(2, -\delta / 3.57) + 199.8 / n + \text{Math.pow}(2, -\delta / 1.61)$

⚠ This code will be executed in your browser. Be sure to only paste trusted content! ⚠

▶ Start

Export / Import

Simulation output

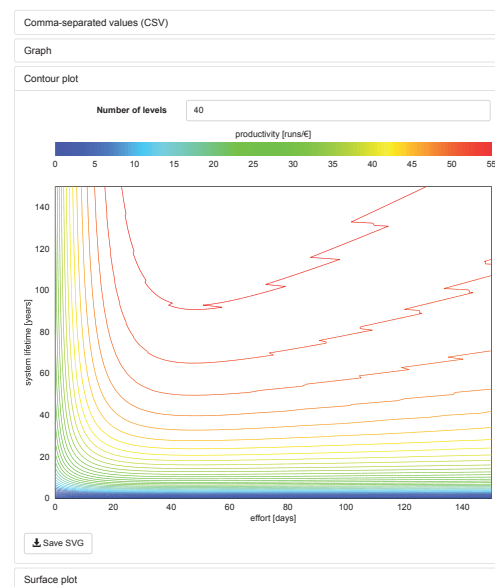


Figure A.2.: Webpage view of the *Aachen HPC Productivity Calculator*. Different input panes and tool tips lead through the definition of values for TCO, application and simulation parameters. Results can be exported to a file or visualized.

A.3. Effort Estimation

A.3.1. COCOMO II Cost Drivers

Table A.7.: Description of effort multipliers (EMs) used in COCOMO II.

EM	Full name	Description
Product		
CPLX	Product Complexity	determined by the type of software, given by ratings for operations in terms of control, computational, device-dependent, data management, and user interface management
DATA	Data Base Size	accounts for effort impacted by large test data requirements, given as ratio of bytes in the database to program SLOC
DOCU	Documentation Match to Life-Cycle Needs	describes the suitability of required documentation to the software life-cycle needs, sample ratings in Tab. 9.5
RELY	Required Software Reliability	captures effect of a software failure, e.g., “slight inconvenience” or “risk to human life”
RUSE	Developed for Reusability	accounts for effort needed to write software components that are intended for reuse in current or future projects, rated by the extent of reuse
Platform		
PVOL	Platform Volatility	rates the stability of the platform, i.e., the hardware, operating system, compilers or data bases, in terms of amount and frequency of changes
STOR	Main Storage Constraint	denotes the needed shares of available main storage
TIME	Execution Time Constraint	categorizes expected execution time needs into percentage ranges of available execution time
Personnel		
ACAP	Analyst Capability	accounts for the analysts’ efficiency, thoroughness, and ability to analyze, design, communicate and cooperate (excluding their level of experience) by giving percentiles
APEX	Applications Experience	captures the applications experience of the project team rated by equivalent level of experience with this type of application given in time spans

Table A.7.: *Continued from previous page*

EM	Full name	Description
LTEX	Language & Tool Experience	represents the experience level in programming language and software tool of the project team in terms of time spans
PCAP	Programmer Capability	focuses on the capability of programmers as a team including their ability, efficiency, thoroughness, and communication skills (excluding their experience levels) given in percentiles
PCON	Personnel Continuity	rates the project's annual personnel turnover in percentages
PLEX	Platform Experience	recognizes the experiences with the underlying technology platform, from graphic user interface over data bases to network, rated in time spans
Project		
SCED	Required Development Schedule	accounts for the effect of the project's schedule compression or expansion as percentage of nominal value, e.g., shorter schedules might need more concurrent developers and thus more costs
SITE	Multisite Development	incorporates site collocation (from on-site to international distribution) and communication support (from phone access to interactive multimedia)
TOOL	Use of Software Tools	represents the effects of tool capability, maturity, and integration depicted as simple edit and code, up to integrated life-cycle management tools

Table A.8.: Description of scale factors (SFs) used in COCOMO II.

SF	Full name	Description
PMAT	Process Maturity	captures the sophistication of the project's development process by either adapting the SEI's Capability Maturity Model (CMM) or by using Key Process Area (KPA) ratings (from "almost always" to "rarely if ever" in terms of, e.g., software project tracking, organization process focus, training program, intergroup coordination or defect prevention)
RESL	Architecture/ Risk Resolution	rates whether the project (manager) attacks risks in advance, given as average of seven characteristics with different scales, example in Tab. 9.5

Table A.8.: Continued from previous page

SF	Full name	Description
PREC	Precedentedness	represents the similarity of the product to previously-developed products and, hence, the familiarity with the product
TEAM	Team Cohesion	accounts for the cooperative or continuous interactions of the project's stakeholders such as users, customers, developers and maintainers
FLEX	Development Flexibility	incorporates the need for software conformance to given requirements and specifications

A.3.2. Ranking of Impact Factors

		p-values											Rank
		CW	KA	KP	PC	TL	PF	HW	CS	AL	PM	EE	sum
effect sizes r	CW		1.00E+00	1.00E+00	1.00E+00	1.00E+00	3.27E-01	1.28E-01	2.49E-01	7.59E-02	1.55E-02	2.02E-04	76
	KA	1.19E-01		1.00E+00	1.00E+00	1.00E+00	1.00E+00	4.16E-01	6.83E-01	1.28E-01	1.05E-03	1.45E-03	83
	KP	8.31E-02	7.71E-02		1.00E+00	1.00E+00	1.00E+00	6.23E-01	1.00E+00	6.06E-01	3.22E-02	5.95E-04	87
	PC	1.91E-01	8.88E-02	1.54E-01		1.00E+00	1.00E+00	6.06E-01	1.00E+00	3.73E-01	2.02E-02	1.03E-04	93
	TL	2.54E-01	1.99E-01	2.31E-01	1.72E-01		1.00E+00	1.00E+00	1.00E+00	1.00E+00	1.99E-01	6.32E-04	112
	PF	3.71E-01	2.64E-01	2.55E-01	2.67E-01	5.64E-02		1.00E+00	1.00E+00	1.00E+00	4.62E-01	5.25E-05	118
	HW	4.20E-01	3.53E-01	3.26E-01	3.27E-01	8.32E-02	5.34E-02		1.00E+00	1.00E+00	6.83E-01	4.62E-03	124
	CS	3.85E-01	3.14E-01	2.45E-01	2.49E-01	1.54E-01	7.99E-02	4.74E-02		1.00E+00	1.00E+00	3.23E-02	131
	AL	4.41E-01	4.21E-01	3.28E-01	3.61E-01	1.63E-01	1.30E-01	9.18E-02	1.48E-02		9.35E-01	3.23E-02	134
	PM	5.07E-01	5.77E-01	4.76E-01	4.95E-01	3.97E-01	3.46E-01	3.17E-01	2.64E-01	2.94E-01		1.00E+00	165
	EE	6.10E-01	5.68E-01	5.89E-01	6.16E-01	5.86E-01	6.23E-01	5.38E-01	4.80E-01	4.77E-01	2.56E-01		197

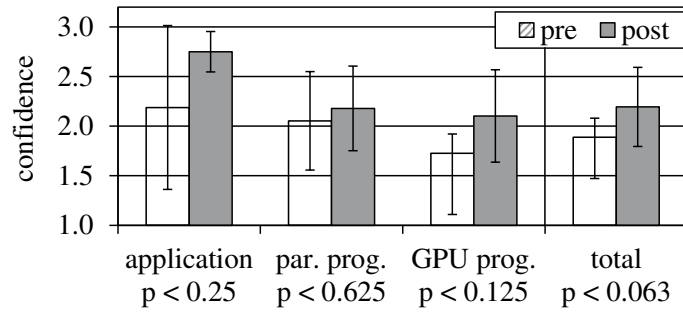
(a) Based on 20 data sets from HPC professionals: Friedman test showed significant difference across factors with $\chi_F^2(10) = 61.264, p < 2.1 \cdot 10^{-9}$.

		p-values											Rank
		KP	CW	PF	HW	PC	KA	TL	AL	EE	CS	PM	sum
effect sizes r	KP		1.00E+00	7.04E-01	6.78E-01	3.35E-01	4.39E-05	2.22E-01	1.32E-03	6.56E-06	3.34E-05	2.25E-05	74
	CW	2.14E-01		1.00E+00	1.00E+00	1.00E+00	1.00E+00	1.00E+00	1.64E-02	4.72E-04	2.53E-05	3.37E-04	99
	PF	2.65E-01	4.35E-02		1.00E+00	1.00E+00	1.00E+00	1.00E+00	2.11E-01	4.29E-05	3.08E-04	4.29E-05	109
	HW	2.76E-01	1.77E-01	6.23E-03		1.00E+00	1.00E+00	1.00E+00	4.07E-01	1.67E-04	4.17E-05	2.53E-05	111
	PC	3.19E-01	1.88E-01	5.39E-02	1.17E-01		1.00E+00	1.00E+00	7.01E-01	1.02E-04	9.92E-05	4.17E-05	118
	KA	6.03E-01	1.99E-01	1.20E-01	2.28E-02	9.32E-02		1.00E+00	7.01E-01	5.03E-04	2.08E-03	1.92E-03	123
	TL	3.41E-01	2.28E-01	1.24E-01	1.42E-01	7.93E-02	2.28E-02		1.00E+00	6.44E-04	2.36E-03	4.10E-04	126
	AL	5.23E-01	4.49E-01	3.44E-01	3.07E-01	2.70E-01	2.71E-01	2.36E-01		1.19E-01	3.02E-03	8.25E-03	162
	EE	6.16E-01	5.48E-01	5.93E-01	5.72E-01	5.79E-01	5.48E-01	5.42E-01	3.73E-01		1.00E+00	1.00E+00	217
	CS	5.97E-01	6.05E-01	5.56E-01	5.96E-01	5.84E-01	5.13E-01	5.07E-01	4.99E-01	1.87E-02		1.00E+00	217
	PM	6.04E-01	5.57E-01	5.93E-01	6.04E-01	5.95E-01	5.15E-01	5.52E-01	4.74E-01	1.75E-01	1.30E-01		228

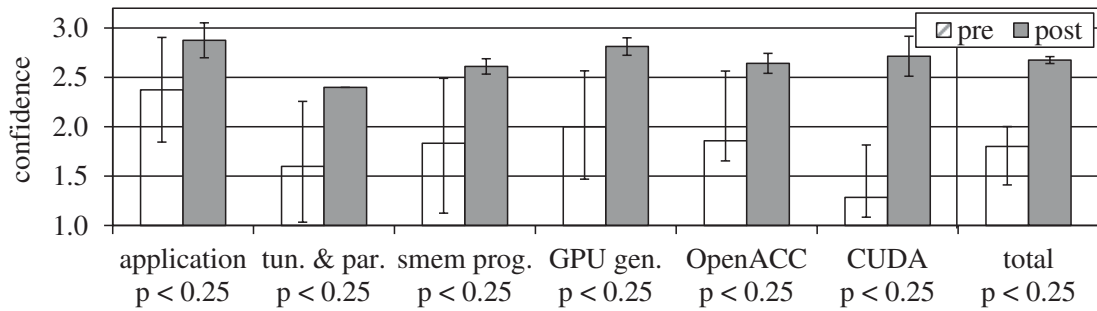
(b) Based on 24 data sets from students: Friedman test showed significant difference across factors with $\chi_F^2(10) = 108.78, p < 2.2 \cdot 10^{-16}$.

Figure A.3.: Results of one-sided signed Wilcoxon rank test with Holm correction. Upper triangle shows p -values (values with $p < 0.05$ in grey). Lower triangle is transposed and shows effect sizes with shade levels $r > 0.1, 0.3$ and 0.5 .

A.3.3. Impact Factor “Pre-Knowledge”



(a) Based on 4 data sets from HPC professionals



(b) Based on 2 data sets from students

Figure A.4.: Means of pre-KS and post-KS results per knowledge question group. Error bars indicate standard deviation.

A.4. Data Collection Tool

A.4.1. Characteristics of EffortLog

An overview of advantages and disadvantages of the three approaches to collect effort data is given in Tab. A.9. In the following, some explanations are given on the assessments of various characteristics.

Keeping manual developer diaries means overhead for the developer in terms of writing down information and remembering effort tracking. In contrast, automatic data collection with Hackystat does not cause any overhead for the developer. However, it requires additional setup effort since tool environments must be instrumented (indicated by (*)). *EffortLog* is easy to setup on all main platforms, and reminds the developer to keep entries. However, it still requires manual logs for annotating categories or noting performance results.

Table A.9.: Comparison of different approaches to collect effort data: assessed as good (+), middle (/), bad (-).

	Manual developer diaries	Automatic data collection (e.g. Hackystat)	EffortLog
Overhead	-	+(*)	/
Context/ interpretation	+	/	+
Information flexibility	+	-	+
Completion	-	+(*)	+
Consistency	-	+	+
Interruptions covered	-	+(*)	/
Accuracy	/	/	?
Control	+	-	+
Socially accepted	+	-	+

Logs from manual and electronic developer diaries can be easily interpreted and put into the correct context through free-form comments and denoted categories. For automatically-collected data this is more challenging because only selected tools are instrumented and having the focus on the tool does not directly mean activity there. Similarly, free-form comments in manual and electronic diaries allow complete flexibility in information, while automatically-collected data are restricted to the given tools.

However, since developers using manual diaries can freely choose on the detail of information, data is often incomplete. Instead, the electronic diary can demand certain data as mandatory and, thus, guarantees a specific level of detail. Tools such as Hackystat also deliver complete data, but only within the given tool set as indicated by (*). The same explanations hold for consistency of collected data.

During development, interruptions may occur in form of colleagues stopping by to talk, receiving and answering e-mails and phone calls, or in form of breaks. Since entries in manual diaries are usually coarse grained, they do not cover these kind of interruptions but focus on ‘real’ work. Tools for automatic data collection may cover these interruptions very well if they are initiated by computer activities, e.g., by activating the e-mail program or by locking the screen during breaks. However, all other interruptions are not covered as indicated by (*). Tracking effort with *EffortLog* represents a tradeoff between these two approaches: If the logging interval is reasonable small, the developer still remembers interruptions and can either exclude the corresponding time or denote further details. Additionally, *EffortLog*’s *Append* functionality allows to log effort away from the computer, e.g., a fruitful discussion in the coffee break.

Due to the coarse-grained character of entries in manual diaries, their accuracy is only moderate. The same holds for the accuracy of automatically-tracked data since it may exclude activity in non-instrumented tools or interruptions as discussed above. *EffortLog* targets at appropriate accuracy by increasing the number of log entries and covering all kinds of activities. Nevertheless, its accuracy has not been proven yet through comparison of logged data to information noted by an external observer, hence, this category is marked with a question mark.

If developers have control on collected data, they and their employers are usually willing to (socially) accept data collection. Control on data is also especially important with respect to German data protection laws. Here, automatically-collected data is transparent to the user, and Hackystat sends it even automatically to a centralized server. In contrast, manual diaries allow full control on data by self-determination on the level of detail. *EffortLog* guarantees full data control by writing logs into local files in a human-readable format. Furthermore, developers have to explicitly hand in collected logs for follow-up data analysis.

A.4.2. User Interface of EffortLog

The figure consists of two side-by-side screenshots of the EffortLog application interface.

(a) Initial configuration: This window is titled "Configuration". It has two main sections: "Project Setup" and "Logging Setup". In "Project Setup", there are fields for "Project title" (containing "myproject"), "Project directory" (containing "C:/MyPrograms" with a "Browse..." button), and "User name" (containing "John Doe"). In "Logging Setup", there are fields for "Output directory" (containing "C:/MyPrograms" with a "Browse..." button), "Log file name" (containing "myproject.json"), and "Logging interval" (a spinner set to "10 min"). Below these sections are two checkboxes: "Did you work before executing this program and want to log it?" and "Do you want to append the work you have done before executing this program to the above specified project?". An "Append..." button is located below the second checkbox. At the bottom are "OK" and "Close" buttons.

(b) Performance-related milestones: This window is titled "Milestone: Working parallel version". It asks the user to "Please specify the following information regarding this milestone:". It contains seven numbered sections: 1. Performance (a dropdown menu set to "Execution time in seconds" and a spinner set to "20.00 sec"); 2. Architecture (a text field containing "2-socket Intel Sandy Bridge @ 2.7 GHz"); 3. No threads, nodes, etc. (a dropdown menu set to "Number of threads" and a spinner set to "16 threads"); 4. Compiler (a text field containing "Intel Compiler 16.0"); 5. Programming model (a text field containing "OpenMP"); 6. Data size (a text field containing "1000^3"); 7. Other comments (a text field containing "General comment on this milestone"). At the bottom are "Skip" and "Finish" buttons.

(a) Initial configuration

(b) Performance-related milestones

Figure A.5.: User interface of *EffortLog*.

(c) HPC activities: categories and comments

Date	Id	Comment	Activity	Duration [min]	Milestone
05-12-2016					
05-13-2016	77	PGI bug in zfsdgb...	Debugging	16	
	78		Tuning	16	
	79		Tuning	16	Working parallel version, Exec. time: 869.7223 sec
	80	Data dependencies...	Debugging	19	
	81		Debugging	17	
	82	Exp with differen g...	Experimenting	16	
	83	Compiler flags	Testing	20	
	84		Break	43	
	85	Compiler flags	Tuning	28	
	86		Testing	22	
	87		Testing	31	
	88		Testing	16	
	89		Experimenting	23	
	90	MPI_Wtime with n...	Debugging	16	
	91		Experimenting	48	
	92	IPA not working in ...	Testing	16	Working parallel version, Exec. time: 863.813 sec
	93		Testing	24	

(d) Log overview for developers. Example taken from *ZFS* tuning activities.Figure A.5.: User interface of *EffortLog*.

A.5. Sensitivity Analysis by Saltelli

Saltelli et al. [STCR04] follow a variance-based sensitivity analysis and propose a numerical procedure for the approximation of the sensitivity coefficients based on Monte Carlo experiments. Given are k model inputs and a base sample size of N that is usually between a few hundreds and a few thousands.

1. Create a random number matrix of size $(N, 2k)$ and split it evenly into two matrices A and B :

$$A = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \dots & x_i^{(1)} & \dots & x_k^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \dots & x_i^{(2)} & \dots & x_k^{(2)} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ x_1^{(N-1)} & x_2^{(N-1)} & \dots & x_i^{(N-1)} & \dots & x_k^{(N-1)} \\ x_1^{(N)} & x_2^{(N)} & \dots & x_i^{(N)} & \dots & x_k^{(N)} \end{bmatrix} \quad (\text{A.1})$$

$$B = \begin{bmatrix} x_{k+1}^{(1)} & x_{k+2}^{(1)} & \dots & x_{k+i}^{(1)} & \dots & x_{2k}^{(1)} \\ x_{k+1}^{(2)} & x_{k+2}^{(2)} & \dots & x_{k+i}^{(2)} & \dots & x_{2k}^{(2)} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ x_{k+1}^{(N-1)} & x_{k+2}^{(N-1)} & \dots & x_{k+i}^{(N-1)} & \dots & x_{2k}^{(N-1)} \\ x_{k+1}^{(N)} & x_{k+2}^{(N)} & \dots & x_{k+i}^{(N)} & \dots & x_{2k}^{(N)} \end{bmatrix} \quad (\text{A.2})$$

2. Create matrix C_i that contains all columns of B except the i th column, instead use the i th column from A

$$C = \begin{bmatrix} x_{k+1}^{(1)} & x_{k+2}^{(1)} & \dots & x_i^{(1)} & \dots & x_{2k}^{(1)} \\ x_{k+1}^{(2)} & x_{k+2}^{(2)} & \dots & x_i^{(2)} & \dots & x_{2k}^{(2)} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ x_{k+1}^{(N-1)} & x_{k+2}^{(N-1)} & \dots & x_i^{(N-1)} & \dots & x_{2k}^{(N-1)} \\ x_{k+1}^{(N)} & x_{k+2}^{(N)} & \dots & x_i^{(N)} & \dots & x_{2k}^{(N)} \end{bmatrix} \quad (\text{A.3})$$

3. Evaluate the model using the three input sample matrices A , B and C_i :

$$y_A = f(A) \quad y_B = f(B) \quad y_{C_i} = f(C_i) \quad (\text{A.4})$$

with y_A , y_B , y_{C_i} of dimension $(N \times 1)$. Thus, only $N(k+2)$ model runs are required instead a cost of N^2 for the brute force method.

4. Estimate the first- and total-effect indices:

$$S_i = \frac{V(E(Y|X_i))}{V(Y)} = \frac{y_A \cdot y_{C_i} - f_0^2}{y_A \cdot y_A - f_0^2} = \frac{(1/N) \sum_{j=1}^N y_A^{(j)} y_{C_i}^{(j)} - f_0^2}{(1/N) \sum_{j=1}^N (y_A^{(j)})^2 - f_0^2} \quad (\text{A.5})$$

where $f_0^2 = \left(\frac{1}{N} \sum_{j=1}^N y_A^{(j)} \right)^2$ represents the mean, and $(\cdot)$ a scalar product.

$$S_{T_i} = 1 - \frac{V(E(Y|X_{\sim i}))}{V(Y)} = 1 - \frac{y_B \cdot y_{C_i} - f_0^2}{y_A \cdot y_A - f_0^2} = 1 - \frac{(1/N) \sum_{j=1}^N y_B^{(j)} y_{C_i}^{(j)} - f_0^2}{(1/N) \sum_{j=1}^N (y_A^{(j)})^2 - f_0^2} \quad (\text{A.6})$$

To gain the input matrices A , B and C_i , the model inputs need to be sampled within their range of variability. Most sampling strategies work on an interval $[0; 1]$. Thus, let Z_i be a parameter with a finite domain $[0; 1]$. If the parameter has another finite domain or an infinite domain, it can be normalized by transformation, e.g., if X_i is a continuous random variable with PDF $f_{X_i}(x)$ and CDF $F_{X_i}(x) = \int_{-\infty}^x f_{X_i}(x')dx'$, then $Z_i = F_{X_i}(X_i)$ or rather

$$X_i = F_{X_i}^{-1}(Z_i). \quad (\text{A.7})$$

If Z_i is uniformly distributed over $[0; 1]$, X_i is distributed following $F_{X_i}(x)$. Thus, each model input needs only be sampled over the range $[0; 1]$, and then the inverse of its CDF can be used to transform the random values back. It can further be distinguished between one-at-a-time (OAT) and all-at-a-time (AAT) sampling methods [PBF⁺16]. As the name suggests, applying the OAT strategy means that one model input is varied at a time while keeping the others fixed. It is typically applied in local SA, but can also be used in GSA. In contrast, in AAT methods, all model inputs are varied at the same time so that their samples contain the direct and joint effects of variation. They are typically used for GSA. Common approaches cover random or quasi-random sampling techniques, e.g., Latin Hypercube sampling (LHS) or Sobol' quasi-random sampling. The VBSA introduced by Saltelli et al. adapts these approaches for a tailored sampling strategy using, e.g., LHS. Here, Latin Hypercube sampling tries to work around the problems of clustering and sample gaps arising in normal random sampling. It stratifies the input PDFs so that points are evenly distributed across a certain number of intervals. More information can be taken from Saltelli et al. [SRA⁺08, pp.76] . Finally, further thoughts should be taken on the sample size N . Especially the convergence of (low or high) sensitivity indices might vary greatly depending on the application. Details can be found in Sarrazin et al. [SPW16].

Statement of Originality

This thesis is based on research and work conducted in the IT Center’s HPC group directed by Prof. Müller and previously Prof. Bischof. Ideas and methods presented in this work strongly benefited from collaborative work and fruitful discussions within the group. Especially, the group leaders Terboven and an Mey shared experiences, provided information and discussed requirements of cluster procurements at RWTH Aachen University. Furthermore, the work with real-world user codes that was often motivated by evaluations of the worth of accelerator-based HPC systems helped to identify challenges and to test methods and solutions. Similarly, my educational activities in terms of supervising student software labs and supervising bachelor and master theses supported my human-subject research. Here, Cramer helped in carrying out classroom experiments, and students provided data, e.g., on their efforts and code performance gains. Some publications were also written in the context of my committee work in the High Performance Group (HPG) of the Standard Performance Evaluation Corporation (SPEC). There, collaboration of numerous academic institutions, national labs and vendors led to the release of the SPEC ACCEL benchmark suite that provides application kernels written in OpenACC and OpenCL, and to its extension by accelerator-type OpenMP codes.

The order of authors on the publications generally reflects the contributions in terms of novel ideas or carried out work. In the following, I provide details on my contributions to these works.

Following publications contributed to my productivity and TCO models presented in Part A and Part B:

- *Accelerators for Technical Computing: Is It Worth the Pain? A TCO Perspective* (Wienke et al. [WaMM13]), *Accelerators, Quo Vadis? Performance vs. Productivity* (Wienke et al. [WTaMM13]), *Modeling the Productivity of HPC Systems on a Computing Center Scale* (Wienke et al. [WIaMM15]): In these publications, we introduced the basics of the single-application productivity model and total costs quantifications for the HPC center at RWTH Aachen University. An Mey provided insights into system-dependent cost factors at RWTH Aachen University. Our joint discussions led to the single-application TCO model that distinguishes node-based and node-type-based

costs. My further contributions were the application-dependent model and analysis, and the projection of TCO results to heterogeneous setups. In follow-up work, my key contribution was the introduction of the productivity figure of merit and the integration of the relationship between effort and performance (here modeled as Pareto Principle). Furthermore, the work covers my idea of including a traditional software cost estimation model such as COCOMO II into the productivity calculation. Iliev included more details on some parameters describing the Pareto Principle, conducted the asymptotic analysis of the model and contributed the application-dependent quantifications from his work with *psOpen*. While paper results were generated with my Matlab implementation of the productivity model, Hahnfeld provided the re-implementation in JavaScript as easy-to-use interface for public availability.

- *Operational Concepts of GPU Systems in HPC Centers: TCO and Productivity* (Schneider et al. [SWM17]), *Application-driven Impact of the Programming Approach on Total Cost of Ownership on the Intel Xeon Phi Architecture — Libraries vs. Intrinsics* (co-advised bachelor thesis of Surudo [Sur14]): Both works illustrate the application of previously-introduced TCO and productivity models to new HPC setups. The first publication was motivated by RWTH procurement considerations for 2016 and 2018 that include operational concepts of GPU-based compute nodes. As part of his student worker activities, Schneider implemented my idea of evaluating these questions with the help of my (single-application based) productivity model and based on an application kernel and a real-world code. In addition, I suggested typical use cases for productivity analysis such as cost of idling hardware and the effect of sharing nodes. In this context, Schneider provided all case study implementations and measurements and conducted the corresponding assessments. In discussions, we refined the different cost models of power consumption during system unavailability. I further co-advised Surudo's bachelor thesis who applied the TCO model for different implementations for the Intel Xeon Phi coprocessor by including his effort logs into the equation, while Cramer supervised the implementation work that covered a CG solver and integration into the iMoose framework.
- *An HPC Application Deployment Model on Azure Cloud for SMEs* (Ding et al. [DaMW⁺13]), *A Study on Today's Cloud Environments for HPC Applications* (Ding et al. [DaMW⁺14]), *Dynamic MPI parallel task scheduling based on a master-worker pattern in cloud computing* (Ding et al. [DWZ15]): These works were mainly conducted by Ding whom I supervised during her internship in Aachen. Here, my main contribution was the integration of a comparison of cloud-based costs of different parallel real-world code implementations that also account for development efforts.

My methodology for development effort estimation introduced in Part C is mostly based on the following publications which also cover the implementation work as foundation for the case study in Part D:

- *Quantifying Productivity — Towards Development Effort Estimation in HPC* (Wienke et al. [WCMS15]), *Development Effort Estimation in HPC* (Wienke et al. [WMSM16]), *Development Effort & Productivity Estimation in HPC* (webpages [Wie16a]): This SC15 poster and SC16 publication is the ground-work for my methodology of effort estimation in HPC. Research on corresponding methods was initiated during my internship at Lawrence Livermore National Laboratory (LLNL), CA, USA, where I benefited from numerous fruitful discussion with LLNL staff, especially Martin Schulz. The novel ideas and analyses in these works were my contributions. This includes the methodology itself, the invention of a performance life-cycle, as well as methods to identify impact factors, the application of knowledge surveys to HPC, and pattern-based analysis of programming model effects. Furthermore, I conducted the corresponding human-subject research in classrooms, hackathons, and other events. I supervised student software labs together with Cramer who provided the initial CG solver and performance-related assignments while my focus was on effort studies. To track these efforts, my former student worker Miller implemented the tool *EffortLog* and, now, keeps on developing it during his own research activities in our group. Furthermore, he contributed first insights into the applicability of COCOMO II to HPC projects that were investigated in detail in further publications (see below). To help jump start the application of my methodology in a broader community, I prepared and put the corresponding material online.
- *A Pattern-Based Comparison of OpenACC and OpenMP for Accelerator Computing* (Wienke et al. [WTBM14]), *Parallel Design Patterns in Comparison on Intel Xeon Phi and NVIDIA GPUs* (co-advised master thesis of Alesja Dammer [Dam14]), *HPC Development Effort Estimation Questionnaire: coMD & CnC* (questionnaire created by Wienke, survey taker Haque): My idea of pattern-based quantification of the impact of the parallel programming model on development time stems back on these publications. First, I applied parallel patterns to show the programmability of two (accelerator) programming models. For that, I profited from Beyer's knowledge of language expressiveness based on his committee work in OpenMP and OpenACC, and an early Cray OpenACC and OpenMP compiler implementation. As follow-up work, Dammer investigated the same parallel patterns on a broad spectrum of programming models and two types of accelerator hardware architectures. I contributed another pattern-based approach for the categorization of applications by providing a questionnaire that was tested by Haque.

- *Software Cost Analysis of GPU-Accelerated Aeroacoustics Simulations in C++ with OpenACC* (Nicolini et al. [NMW⁺16]), *Software Cost Estimation of GPU-accelerated Aeroacoustic Simulations with OpenACC* (co-advised bachelor thesis of Nicolini [Nic15]), *Applicability of the Software Cost Model COCOMO II to HPC Projects* (Miller et al. [MWSL⁺17]), *Software Cost Estimation of GPU-accelerated Aeroacoustic Simulations with OpenACC* (co-advised master thesis of Miller [Mil16]): I supervised two student theses and provided corresponding ideas and concepts with respect to accelerator programming, effort-performance evaluations and the application of the COCOMO model, while activities with respect to *ZFS* — the application under investigation — were supervised by Schlottke-Lakemper from the Institute of Aerodynamics. The first parallel OpenACC version of *ZFS* was implemented by Nicolini as part of his bachelor thesis. He also described required code transformations and a performance analysis in a follow-up publication. In his bachelor thesis, he also included effort logs and first investigations of COCOMO’s applicability to this project. Building upon this initial OpenACC port, Miller highly tuned *ZFS* for GPUs while applying *EffortLog* for effort-performance evaluations as part of his master thesis. Furthermore, he set up COCOMO parameters for this *ZFS* tuning project and compared it to his reported efforts. Taking up my initial ideas to account for modified and existing kernel code lines, he interpreted this as part of COCOMO’s Reuse Model (ASLOC). His applicability study of COCOMO to *ZFS* was also part of two publications. Another task of his master thesis comprised the analysis of student effort-performance pairs collected by me in software labs over several years. Based on our discussions, he investigated the impact of developer experiences (in terms of students’ grades) by quantifying turning points in the performance life-cycles. I further contributed to our follow-up publications by setting up COCOMO parameters for the software lab data and compared students’ actual efforts to COCOMO estimations. Moreover, I established and assessed the model and its conditions for an uncertainty and sensitivity analysis of student lab and *ZFS* data with corresponding analysis. The conclusions of applicability of COCOMO II to HPC projects were mostly derived by discussions of Miller and me.

Applicability of models and methods is shown in numerous case studies with the help of application kernels or real-world HPC frameworks. The following works incorporate groundwork for these investigations, e.g., in terms of effort evaluation.

- *Simulation of bevel gear cutting with GPGPUs — performance and productivity* (Wienke et al. [WPM⁺11]), *Produktivität bei der plattformübergreifenden Beschleunigung der Spannungsdickenberechnung für das Kegelradfräsen mit OpenCL* (co-advised bachelor theses of Plotnikov [Plo10]), *OpenACC — First Experiences with Real-World Applications* (Wienke et al. [WSTaM12]), A

Study of Productivity and Performance of Modern Vector Processors (co-advised bachelor thesis of Springer [Spr12]), *Comparing the Programmability of Accelerators: OpenMP 4.0 vs. OpenACC* (co-advised bachelor thesis of Neumann [Neu13]): These publications and co-advised students' bachelor theses cover experiences from (mostly accelerator-based) parallelization activities of real-world application and respective effort evaluations. Here, students mainly provided different code implementations, while I focused on the programmability and usability of different parallel programming models and hardware architectures (usually in terms of lines of code and required development time). In detail, Plotnikov provided an OpenCL and CUDA implementation of *KegelSpan* and evaluated its performance on several platforms, while I provided a tuned PGI Accelerator and OpenACC version of the code. Highly-tuned versions of the real-world application *NINA* with OpenMP (for CPUs), OpenCL and CUDA were created by Springer who also provided the number of required modified lines of code and a descriptive analysis of programming productivity in his thesis. Again, I focused on directive-based accelerator programming with my implementations of *NINA* in PGI Accelerator and OpenACC. Neumann was asked to port the OpenACC versions of *KegelSpan* and *NINA* to OpenMP-target capabilities while tracking corresponding development efforts. Since our apprentice Hahnfeld obtained an OpenMP-target *NINA* version with higher performance, I used his implementation for further studies.

- *Assessing the Performance of OpenMP Programs on the Intel Xeon Phi* (Schmidl et al. [SCW⁺13]), *Performance Portability Analysis for Real-Time Simulations of Smoke Propagation using OpenACC* (Küstters et al. [KWA17]), *Towards an accurate simulation of the crystallisation process in injection moulded plastic components by hybrid parallelisation* (Wienke et al. [WSD⁺14]): While most of my works focus on effort evaluations, few publications also emphasize the performance aspect of real-world codes. Based on Springer's work, I contributed a *NINA* code version using OpenMP and Intel's Language Extensions for Offload (LEO) that obtained good performance and scalability on an Intel Xeon Phi coprocessor. In addition, Dammer and I worked on a task-driven MPI+OpenMP version of *SphaeroSim* that provides good scalability on traditional compute nodes. In the context of a seed fund project, we also investigated the MPI+OpenMP performance of *SphaeroSim* on an Intel Xeon Phi coprocessor and found several issues that hinder the efficient operation of this huge real-world C++ code on this architecture. I further mentored the OpenACC parallelization of the real-world code *JuROR* in a GPU hackathon, and led an analysis of performance portability on several architectures.
- *SPEC ACCEL: A Standard Application Suite for Measuring Hardware Accelerator Performance* (Juckeland et al. [JBC⁺14]), *From Describing to Pre-*

scribing Parallelism: Translating the SPEC ACCEL OpenACC Suite to OpenMP Target Directives (Juckeland et al. [JHJ⁺16]): In the context of my SPEC HPG work, the committee published results from the creation of the SPEC ACCEL benchmark suite, and experiences from translating OpenACC application kernels to OpenMP accelerator code. I contributed to these by actively engaging in precedent discussions on different implementations and performance impacts using my experiences from other real-world code implementations with OpenACC and OpenMP. Furthermore, I put our OpenACC-to-OpenMP code ports into the context of other related works.

The ever increasing demands for computational power tighten the reigns on available budgets. Constraints on expenses for acquisition and electrical power must be met, yielding continuously-increasing hardware and software complexity that application developers have to deal with. Thus, informed decision making on how to invest available budgets is more important than ever. Especially for HPC procurements, a quantitative metric helps to predict the cost effectiveness of an HPC center. However, prevailing metrics such as Linpack Flop/s, or Flop/s per watt only pick up part of the picture that HPC centers are concerned with, i.e., expenses for hardware, software, maintenance, infrastructure, energy, programming effort, as well as the value that researchers get from the HPC system in terms of scientific output.

In this work, I set up methodologies to support the HPC procurement process of German HPC centers. I model cost effectiveness of HPC centers as a productivity figure of merit by defining a ratio of scientific outcome generated over the lifetime of the HPC system to its total cost of ownership (TCO). Scientific outcome is further defined as number of scientific-application runs to embrace the multi-job nature of an HPC system in a meaningful way. I investigate the predictability of the model's parameters and show their robustness towards errors in various real-world HPC setups. The TCO component of my productivity model covers one-time and annual expenses and distinguishes between node-based and node-type-based costs. Costs for development efforts needed to parallelize, port and tune simulation codes to efficiently exploit HPC systems must also be part of a sound productivity model. Since software cost models from mainstream software engineering do not focus on the laborious task of squeezing out the last percentage points of runtime performance, I introduce a methodology to estimate corresponding HPC efforts. It is based on so-called performance life-cycles that model the relationship of performance and effort required to achieve that performance. My methodology further covers the identification and quantification of various impact factors on HPC development effort and provides methods to collect required data sets from human subjects for statistically reliable results. Finally, I present the applicability of my methodologies and models in a case study that covers a real-world application from aeroacoustics simulation.

ISBN 978-3-86359-572-2



9 783863 595722