Adaptive Parallel Communications for Large-Scale Computational Fluid Dynamics

A thesis accepted by the Faculty of Energy Technology, Process Engineering and Biological Engineering of the Universität Stuttgart in partial fulfilment of the requirements for the degree of Doctor of Engineering Sciences (Dr. Ing.)

by

Katharina Benkert

from Augsburg

Main-referee:	Prof. Dr. M. Resch
1. Co-referee:	Assoc. Prof. Dr. E. Gabriel
2. Co-referee:	Prof. Dr. S. Roller
Date of defence:	23.9.2011

Institut für Höchstleistungsrechnen der Universität Stuttgart

2011

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über
http://dnb.d-nb.de abrufbar.

ISBN 978-3-8439-0231-1

© Verlag Dr. Hut, München 2011
Sternstr. 18, 80538 München
Tel.: 089/66060798
www.dr.hut-verlag.de

Die Informationen in diesem Buch wurden mit großer Sorgfalt erarbeitet. Dennoch können Fehler nicht vollständig ausgeschlossen werden. Verlag, Autoren und ggf. Übersetzer übernehmen keine juristische Verantwortung oder irgendeine Haftung für eventuell verbliebene fehlerhafte Angaben und deren Folgen.

Alle Rechte, auch die des auszugsweisen Nachdrucks, der Vervielfältigung und Verbreitung in besonderen Verfahren wie fotomechanischer Nachdruck, Fotokopie, Mikrokopie, elektronische Datenaufzeichnung einschließlich Speicherung und Übertragung auf weitere Datenträger sowie Übersetzung in andere Sprachen, behält sich der Autor vor.

1. Auflage 2011

In memory of my mother
Ursula "Wu" Benkert

Contents

1	**Introduction**	**1**
1.1	High-Performance Computing for Simulations	1
1.2	Requirements for HPC	2
1.3	The Difficulties and Drawbacks of Code Tuning	3
1.4	The Impact on Scientific Simulations	4
1.5	Automatic performance tuning	5
1.6	Outline	6
2	**Fundamentals**	**7**
2.1	The governing equations	8
2.2	Finite Volume Methods	9
2.2.1	Discretization in space	9
2.2.2	Discretization in time	12
2.2.3	Boundary conditions	13
2.2.4	Higher-order schemes	13
2.2.5	Algorithm and Parallelization	15
2.3	Spectral Methods	16
2.3.1	Derivation of the spectral equations	17
2.3.2	Algorithm and Parallelization	18
2.4	Application Test Cases	19
2.4.1	Euler3D	19
2.4.2	The NAS FT Benchmark	19
2.5	HPC Systems	22
3	**Automatic Performance Tuning**	**25**
3.1	Historic Overview	25
3.1.1	The Beginnings	25
3.1.2	Formalization efforts	27
3.1.3	Linear algebra	27
3.1.4	Solvers	28
3.1.5	Automatic performance tuning for MPI communications	29
3.2	Characteristics of automatic tuning systems	30
3.2.1	When to Tune	30
3.2.2	How to Tune	31

3.2.2.1 Heuristic modelling 31
3.2.2.2 Empirical Evaluation 32
3.2.2.3 Advantages and Disadvantages of Heuristic Modelling and Empirical Evaluation 33
3.3 The Abstract Data and Communication Library (ADCL) 33
3.3.1 Using ADCL 36
3.3.2 Mode of operation 39
3.3.3 Overheads and Countermeasures: A Sophisticated Selection Logic and Historic Learning 40
3.4 Previously Missing Features and Contributions of This Work . . 42
3.4.1 Area of Application 42
3.4.2 Outlier Handling 43
3.4.3 Reliable Performance Measurements 44
3.4.4 Conclusions 44

4 Extending the Area of Application of ADCL 47
4.1 Semantics of new ADCL interfaces 47
4.1.1 The ADCL vector-map object 48
4.1.2 Extension of the ADCL Interfaces 49
4.1.3 The new function sets 51
4.2 Performance Evaluation 53
4.2.1 Integration of ADCL 53
4.2.2 Setup 53
4.2.3 Results 55
4.3 Conclusion 56

5 Outlier Analysis and Evaluation of Different Decision Algorithms 59
5.1 Outliers in Performance Data 59
5.2 Techniques for Performance Data Evaluation 61
5.2.1 Heuristic approach 62
5.2.2 Standard Interquartile Range Method 63
5.2.3 Cluster Analysis 64
5.2.4 Robust statistics 64
5.2.5 Complexity estimates 66
5.3 Performance Evaluation 67
5.3.1 Integration of the ADCL 67
5.3.2 Setup 68
5.3.3 Results 70
5.3.3.1 Characterizing the performance of codelets 71
5.3.3.2 Potential Performance Benefits of ADCL 73
5.3.3.3 Statistical Data Analysis 78
5.3.3.4 Performance of ADCL 85

5.4 Conclusion 85

6 Timing Techniques for Collective Communications 89
6.1 Timing Techniques to Generate Performance Data 89
6.2 Technical Concept of the ADCL Timer Object 92
6.2.1 Case 1: Optimizing One Request 92
6.2.2 Case 2: Optimizing Multiple Requests 94
6.2.3 Integration of the Timer Object 95
6.3 Performance Evaluation 98
6.3.1 Ranking the codelets from best to worst 98
6.3.2 Assessment of the timing techniques 101
6.4 Conclusion 106

7 Conclusions and Outlook 107

Bibliography 111

Glossary 121

Nomenclature

b	Bound for heuristic outlier filtering approach
c	Codelet
c^{meta}	Meta codelet
$\mathcal{C}$	Condition for empirical data point to fulfill to be not an outlier
e	Internal energy
$e_{\min}(c,d)$	Minimum error concerning the ranking of two codelets c and d according to their performance
$e_{\text{avg}}(c,d)$	Average error concerning the ranking of two codelets c and d according to their performance
$e_{\max}(c,d)$	Maximum error concerning the ranking of two codelets c and d according to their performance
η	Dynamic viscosity
E	Total energy
$g_{\text{avg}}(k)$	percentage gain/loss when comparing the averaged verification runtimes of the code version with an ADCL codelet k to those of the original code without ADCL
$I(c)$	Instability of a codelet c
K	Problem class for NAS Parallel Benchmark
L	Characteristic linear dimension
m	Measurement
μ	Mean
$\hat{\mu}$	Estimated mean

$\hat{\mu}_f$	Estimated mean based on the set of filtered empirical data S_f
$\hat{\mu}^i_M(c)$	Estimated mean for codelet c of the i-th verification run using timing method M
M	Timing method
nD	n dimensional
n_c	Number of codelets for a communication pattern
n_f	Cardinality of S_f
n_m	Number of measurements executed for one codelet during the optimization process
n_o	Number of outliers
$n_o^{\max}$	Maximum number of outliers for heuristic outlier filtering approach
n_p	Number of processes
n_r	Number of verification runs
n_s	Synchronization interval for *timer_multistep*
$\mathcal{N}(\mu, \sigma^2)$	Normal distribution with mean μ and variance σ^2
$o(c, d)$	Overlap of a codelet c with a codelet d
O	Landau symbol
p	Parallel process
P	Pressure
$P_v^{\min}(c)$	Minimum percentage overhead of a codelet c compared to the best codelet based on the verification runs
$P_v^{\text{avg}}(c)$	Average percentage overhead of a codelet c compared to the best codelet based on the verification runs
$P_v^{\max}(c)$	Maximum percentage overhead of a codelet c compared to the best codelet based on the verification runs
$P_M^c(i)$	Percentage overhead of a codelet c compared to the best codelet in the i-th verification run using timing method M

Q	Quartile
ρ	Fluid mass density
Re	Reynolds number
s	Sample standard deviation
σ^2	Variance
S_c	Set of ADCL codelets for one communication pattern
S_c^{meta}	Set of meta codelets
S_f	Set of filtered empirical data
sgn	Signum function
t	Execution time
$t\,\{\mu, \psi, \nu\}$	t-distribution with location parameter μ, scale parameter ψ and ν degrees of freedom.
$\hat{t}_h$	Estimated average execution time of a codelet for the heuristic approach
t_i^k	Execution time of a code version k in the i-th verification run
t_k^{min}	Minimum execution time of a code version k over all verification run
t_k^{avg}	Average execution time of a code version k over all verification run
t_k^{max}	Maximum execution time of a code version k over all verification run
$t_v^i(c)$	Execution time of a codelet c in the i-th verification run
$t_v^{\text{min}}(c)$	Minimum execution time of a codelet c over all verification runs
$t_v^{\text{avg}}(c)$	Average execution time of a codelet c over all verification runs
$t_v^{\text{max}}(c)$	Maximum execution time of a codelet c over all verification runs
u	Fluid velocity
v	Mean fluid velocity

$w_{\min}(c,d)$	Minimum percentage overhead / gain between two codelets c and d based on the verification runs taking into account the overlap $o(c,d)$
$w_{\text{avg}}(c,d)$	Average percentage overhead / gain between two codelets c and d based on the verification runs taking into account the overlap $o(c,d)$
$w_{\max}(c,d)$	Maximum percentage overhead / gain between two codelets c and d based on the verification runs taking into account the overlap $o(c,d)$
AEOS	Automated Empirical Optimization of Software
API	Application Programming Interface
B	Simple timing method using synchronization (*barrier*)
CFD	Computational Fluid Dynamics
DFT	Discrete Fourier Transform
DNS	Direct Numerical Simulation
FD	Finite Differences
FFT	Fast Fourier Transform
Flop	Floating Point Operation per Second
GB	Gigabyte
HLRS	High Performance Computing Center Stuttgart
HPC	High Performance Computing
impi	Intel MPI
IQR	Interquartile range
LFC	LAPACK For Clusters
MAD	Median of Absolute Deviations
MB	Megabyte
ML	Maximum Likelihood
MLE	Maximum Likelihood Estimation

MPI	Message Passing Interface
NB	Simple timing method without synchronization (*nobarrier*)
NSE	Navier-Stokes Equations
ODE	Ordinary Differential Equation
ompi	OpenMPI
PDE	Partial Differential Equation
SANS	Self-Adapting Numerical Software
SMP	Symmetric Multi-Processing
T	Timing method *timer* based on the timer object
TFQMR	Transpose Free Quasi-Minimal-Residual
TM	Timing method *timer_multistep* based on the timer object with infrequent synchronizations

Civilization advances by extending the number of important operations
which we can perform without thinking about them.
Alfred North Whitehead

Abstract

Nowadays, simulation methods in engineering are recognized as equally important as the traditional fields of theory and experiment. They reduce or replace expensive experiments and allow simulations of physical processes which are impossible to investigate otherwise. Because of their complexity, parallel computations are necessary, commonly using the Message Passing Interface (MPI) in distributed memory environments.

One of the main obstacles end-users are facing when performing large-scale computations is the choice between performance and portability. The performance of MPI communications depends on various factors such as network interconnect, MPI implementation, process placement and message sizes. The complexity of the MPI standard exceeds the average end-user's knowledge. A possible solution are empirical optimization libraries which offer a rich set of codelets, i.e. implementations, for a particular problem as well as methods to detect the best-performing one. This allows for tuning applications at install- or runtime without special knowledge or intervention of the end-user.

In this work, two large-scale Computational Fluid Dynamics (CFD) applications are optimized using the empirical auto-tuning library ADCL, the finite-volume code Euler3D and a Fast Fourier Transforms benchmark as kernel of spectral CFD methods. For this, ADCL is extended beyond neighborhood communication patterns to allow tuning of collective communications. It is shown that ADCL can shorten runtimes more than 30% for the chosen test cases.

As empirical optimization libraries base their choice of the best-performing codelet on empirical data, this work also investigates two fundamental problems that are associated with collecting this data and its evaluation. Firstly, the empirical data is obtained by measuring the execution times of various codelets where the measurement method greatly influences its informational value. Secondly, the evaluation of the data is encumbered by unpredictable variations in execution time which frequently occur in a parallel setting resulting in data points which differ greatly from the observed average, so-called *outliers*. In this thesis, recommendations are given on how runtime tuning in a parallel environment needs to be carried out to get optimal results.

Based on the results presented in this work, the library ADCL now possesses the means to tune MPI communications in most application scenarios and uses a sound empirical framework to choose the best-performing codelet in MPI

parallel simulations. Optimization of collectives as well as code which overlaps communication and computation is now possible. This makes the performance of MPI communications portable, i.e. the communications perform well on different machines without spending anew time on optimizations, and together with the easy use of the library, reduces the complexity for the user.

Zusammenfassung

In den Ingenieurwissenschaften wird heutzutage Simulationsmethoden die gleiche Bedeutung zugemessen wie den herkömmlichen Bereichen Theorie und Experiment. Sie reduzieren oder ersetzen teuere Experimente und erlauben Computersimulationen physikalischer Prozesse, die sonst nicht untersuchbar wären. Wegen ihrer Komplexität sind parallele Berechnungen notwendig, die normalerweise auf Systemen mit verteiltem Speicher unter Verwendung von Message Passing Interface (MPI) ausgeführt werden.

Eines der Hauptprobleme, dem die Benutzer begegnen, wenn sie großskalige Berechnungen durchführen, ist die Wahl zwischen schneller Ausführung und Portabilität. Die Eigenschaften der MPI Kommunikationen hängen von verschiedenen Faktoren ab, beispielsweise vom Netzwerktyp, der MPI Implementierung, auf welchen Knoten die Berechnung ausgeführt wird oder den Nachrichtenlängen. Die Komplexität des MPI Standards ist größer als das Wissen eines durchschnittlichen Benutzers. Eine mögliche Lösung sind empirische Optimierungsbibliotheken, die eine große Anzahl an möglichen Implementierungen, sogenannten Codelets, für ein spezielles Problem zur Verfügung stellen sowie Methoden um das Beste darunter auszuwählen. Dies erlaubt es, Anwendungen zur Installations- oder Laufzeit zu optimieren ohne dass besondere Kenntnisse oder ein Eingreifen des Benutzers notwendig sind.

In dieser Arbeit wird die empirische Optimierungsbibliothek ADCL verwendet, um zwei großskalige Strömungssimulationen zu optimieren, den Finite-Volumen-Code Euler3D und eine Schnelle Fourier-Transformation (FFT), die den Kernel spektraler Strömungssimulationsmethoden bildet. Dafür wird ADCL erweitert, so dass nicht mehr nur Nachbarschaftskommunikationen sondern auch kollektive Kommunikationen optimiert werden können. Es wird gezeigt, dass ADCL die Laufzeit der Simulationen für die gewählten Beispiele um bis zu 30% verkürzt.

Da empirische Optimierungsbibiliotheken die Wahl des schnellsten Codelets basierend auf den empirischen Daten treffen, untersucht diese Arbeit auch zwei grundsätzliche Probleme, die mit der Erzeugung und dem Auswerten der Daten in Verbindung stehen. Erstens werden die empirischen Daten gewonnen, indem die Ausführungszeiten der verschiedenen Codelets bestimmt werden. Dabei beeinflusst die Methode, wie gemessen wird, zu einem großen Maß den Informationsgehalt der Daten. Zweitens wird die Auswertung der Daten

durch unvorhersehbare Einflüsse auf die Ausführungszeit erschwert. Diese Phänomene treten oft in parallelen Anwendungen auf und bewirken, dass Datenpunkte, sogenannte Ausreisser, sehr von dem beobachteten Durchschnitt abweichen. In dieser Dissertation werden Empfehlungen gegeben wie Optimierungen zur Laufzeit in einer parallelen Umgebung ausgeführt werden sollten um optimale Resultate zu erzielen.

Basierend auf den Ergebnissen dieser Arbeit verfügt die Bibliothek ADCL jetzt über die Mittel, MPI Kommunikationen in den meisten Anwendungsszenarien zu optimieren und besitzt ein solides empirisches Rahmenwerk, um das schnellste Codelet in MPI parallelen Simulationen auszuwählen. Die Optimierung von kollektiven MPI Operationen sowie Programmcode, bei dem sich Kommunikation und Berechnung überlappen, ist jetzt möglich. Dies macht die Performance von MPI Kommunikationen portabel, d.h. die Kommunikationen sind auf unterschiedlichen Maschinen schnell ohne dass man erneut Zeit für Optimierungen aufwenden müsste. Dies reduziert zusammen mit der einfachen Benutzung der Bibliothek die Komplexität für den Benutzer erheblich.

1 Introduction

With the development of computers, it has become possible to replace experiments that are either too dangerous, too expensive or plainly impossible to carry out with simulations based on theoretical models.

Nowadays, simulations have become affordable and are integrated in everyday research and industrial activities. The most common goals are to reduce costs during product design or to improve existing products. The areas of applications are numerous: in turbine design, simulations replace expensive prototypes and help to achieve optimum efficiency [1, 2]. Industrial coating processes are optimized—without material loss—to yield a high-quality, thin and even coating [3]. Simulations improve injection mechanisms [4, 5] and air-conditioning [6, 7] in cars or help reducing noise and CO_2 emissions by optimizing airfoil design [8].

1.1 High-Performance Computing for Simulations

Computationally intensive simulations require leading edge technology, i.e. High Performance Computing (HPC) systems. This applies to emerging high-resolution simulations (e.g. for earth quake models to predict the damages as accurately as current technology allows [9]), multi-physics applications that integrate different physical phenomena into a single simulation (e.g. combining gas flow and particle transport to model lunar probes [10]), multi-scale simulations encompassing different resolutions of a domain (e.g. a regional model of the Agulhas current system coupled with one global coarse resolution model to predict the influence on the Atlantic meridional overturning circulation [11] on the global circulation variability) or parameter studies such as power plant simulation with varying burners and fuels quality [12].

An especially interesting subject in engineering which falls into the category of computationally intensive simulations are Computational Fluid Dynamics (CFD). CFD is an expression used for numerical methods and algorithms devised to resolve fluid flows. Viscid flows are described analytically by the Navier-Stokes equations (NSEs).

Traditional CFD methods like the often employed Finite Volume method discretize and solve this equation directly to obtain hydrodynamic variables like density or the fluid velocity. Solving the NSEs directly without averaging or approximations other than the numerical error is called a direct numerical simulation (DNS). It is the most accurate approach to turbulence simulation and a crucial tool for research carried out at the Institute of Aero- and Gasdynamics of the University of Stuttgart. It is also still one of the most challenging problems of HPC since computational power and a large main memory are necessary to complete a high resolution simulation in an adequate time frame. The first direct simulation of homogeneous, isotropic turbulence was done by Orszag and Patterson [13] on a 32^3 grid using a spectral Galerkin method in 1972. In 2006, Yokokawa et al. [14] did an ground-breaking and in size unpreceded DNS of incompressible turbulence at large Reynolds numbers. It ran on 512 nodes of the Earth Simulator with 2048^3 (double precision) or 4096^3 (single precision) grid points and a sustained performance of up to 16.4 TFlops.

1.2 Requirements for HPC

Serial processor performance is reaching its limits: the power per chip cannot increase further (power wall), frequency scaling has reached its physical limits and the introduction of further logic to improve Instruction Level Parallelism (ILP) does not pay off (ILP wall) and the gap between CPU cycle time and memory access latencies, the memory wall [15, 16, 17], increases. A way out provide multi- and many-cores architectures, accelerators and the use of more and more processes. The exploitation of this parallelism needs explicit instructions from the user as compiler-support is not available or in its infancy. The dominating parallel programming model is message-passing usually following the Message Passing Interface (MPI) [18] specification. There are two types of communication patterns: point-to-point and collective communications.
A *point-to-point communication* describes the directed message exchange between a sending and a receiving process. It occurs frequently as part of a neighborhood communication pattern, where one process exchanges data with its next neighbors. This pattern is found in stencil computations, e.g. in Lattice Boltzmann methods or finite difference schemes, or originates from physical properties, such as in molecular dynamics where particles do not hop from one end of the computational domain to the other.
A *collective communication*, or short *collective*, is defined as communication that involves a group or groups of processes. The latter is/are specified via a MPI communicator. Apart from two exceptions (`MPI_SCAN, MPI_EXSCAN`), the collective operations can be categorized into communications of type all-to-all,

all-to-one and one-to-all. In case of one group or an so-called *intra-communicator*, data is exchanged within the group following the types of communications just mentioned. For multiple groups of processes, i.e. if an *inter-communicator* is present, data from one group of processes is provided to the other groups. In most cases, collectives include all processes using the predefined communicator `MPI_COMM_WORLD`. Collective operations are employed for example for vector norms or scalar products.

An important property of a parallel program is its good scalability. *Scalability* describes the ability of software to run efficiently on any number of processors. To allow simulation software to scale to a high number of processes, extensive code tuning is required. This is especially true for communication operations as their impact increases with the number of processes used.

1.3 The Difficulties and Drawbacks of Code Tuning

Tuning code is one of the main challenges of HPC simulations. General Purpose Graphics Processing Units (GPGPUs) and cell processors with hundreds of GFlop/s, clusters with hundreds of thousands of processors and dropping prices for hardware suggest that running complex simulations in short times is easily feasible. However, without extensive code tuning it is not possible to exploit the capabilities of HPC systems.

Highly trained personnel often has to introduce hand-coded optimizations into the simulation software. This process is time-consuming, error-prone and the results are in most cases not portable, i.e. the tuned code does not perform well on other systems. This contradicts with fundamental ideas of software engineering to obtain high quality, re-usable, re-engineerable, maintainable and testable scientific software. Effects of tuning are hard to judge because the software environment (compilers, libraries, ...) has to be taken into account. As a first measure, computer manufactures and independent software vendors (ISV) provided highly optimized libraries for common computationally intensive parts such as linear algebra [19, 20, 21, 22] operations and Fast Fourier Transforms (FFTs) [23]. With a steady increase and diversification of available computer architectures, this task has become more and more difficult. Optimizations become obsolete or counterproductive over time as technology advances. This is especially true for current and future chip designs such as modern multi-core processors which have a significantly different layout depending on the manufacturer. This concerns the organization and connectivity of the cores, different cache hierarchies and I/O capabilities of the processors.

When moving from single CPU tunings to performance optimization for shared or distributed memory systems, tuning complexity increases dramatically due to the dependency on the number of processes. Run-time effects such as concurrent network traffic or operating system (OS) jitter increase the variance of performance. Many more considerations regarding parallel efficiency have to be taken into account. This includes varying network hardware, device drivers and interconnects, different MPI libraries, latencies when accessing memory of remote MPI processes as well as multiple possibilities to employ MPI functionality.

MPI, although unrivaled for distributed memory parallelization, is often playfully termed as assembler of parallel programming as it is difficult to program and offers numerous possibilities for (untraceable) programming errors. In addition, MPI requires from the user an in-depth knowledge of the voluminous standard, a knowledge which is often not existing: as result of a six-month survey [24], it was noted that end-users do not exploit MPI's possibilities to their full extent, although considerable time, on average 13.7% of the application CPU time, are spent in MPI functions offering plenty of possibilities for optimization. More than 99% of the MPI runtime was spent in only 19 out of the 176 MPI routines. The pack and unpack functions are rarely employed. Derived data types are seldomly utilized outside the development phase. Non-blocking point-to-point communication is less frequently used than blocking communications. From a user's perspective, it might be worthwhile to know that there exist 20 possible variations of how to realize a neighborhood communication pattern, but implementing all of them and finding out which of them is the fastest for each HPC system, MPI library, compiler, etc. is out of question. What is needed is an abstraction which allows to specify which data to communicate and where to send it.

1.4 The Impact on Scientific Simulations

Figure 1.1 shows the typical development cycle of scientific simulation software. New ideas and models are implemented and tested on a standard PC or workstation first. As research advances and the complexity of test cases increases computational power becomes the limiting factor. The demand emerges to move up the so-called performance pyramid to larger systems, i.e. to clusters, regional computing centers and ultimately to an HPC center. With the knowledge gained from the HPC simulations and the on-going improvement of computer hardware, calculations become less costly over time and can be moved back onto smaller-scale machines. When the findings lead to revised models or new algorithms the development cycle is closed. Each transition from

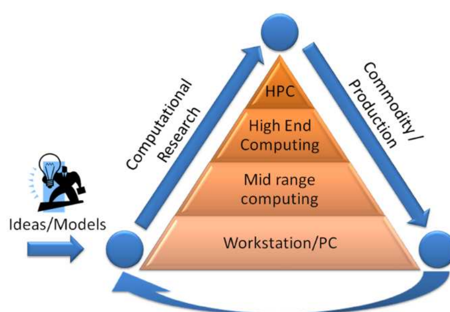


Figure 1.1: Software development cycle for HPC simulations [25].

one level of the performance pyramid to another (cf. fig. 1.1) normally entails a certain amount of code modifications. Examples are the introduction of a distributed memory parallelization when moving from workstations to mid-range computing and optimization for special hardware features when deploying the software on HPC systems. A seamless transition benefits the researchers as it allows them to focus on their research instead of investing considerable time in dealing with software issues. However in practice, such a transition is often difficult. Hard- and software stack differ greatly between levels and between computing centers on the same level, e.g., different processors and networks, the availability of accelerators, diverse compilers, device drivers and communication libraries.

In conclusion, computational intensive simulations require HPC which in turn can only be used effectively if the simulation software is scalable and optimized for the specific soft- and hardware environment. This code tuning is challenging, time-consuming and requires special training. Optimization is however system specific and thus needs to be carried out anew when moving from one system to another. Otherwise performance penalties are almost inevitable. The solution to overcome this dilemma are automatic tuning mechanisms that adapt the software to the underlying environment without human involvement.

1.5 Automatic performance tuning

Automatic performance tuning is an area of research defined by one of the fundamental questions in computing: how to provide—with reasonable effort—efficient code on a wide variety of computer architectures steadily increasing in complexity and diversity. Or, in other words, how to obtain for a kernel (computational kernel, communication pattern) platform-independently an equal or superior performance compared to hand-tuned code. A *kernel* in this

context is a confined entity of a computer program of special importance, e.g. the central part of the algorithm or a computationally expensive well-defined set of operations. An alternate goal of automatic performance tuning is to hide complexity from the user for example by providing simplified interfaces.

Following the ideas of Whaley [26], an *automatic tuning system* encapsulates performance-critical routines (kernels) into a library and offers various codelets for each kernel as well as timing routines and search techniques. A *codelet* is a piece of code that provides one way to solve a particular problem. After the performance of the codelets has been assessed during a search phase, the system decides on a codelet judged best-performing and uses it during the simulation. An *empirical search-based tuning system* bases its decision on empirical data, i.e. measurements of execution times.

Auto-tuning MPI communications has its challenges. Depending on various factors, some measured executions times may not fit into the overall pattern and require special treatment. The informational value of the measurements has to be high as one is interested in a short search phase with only a few measurements per codelet. The Abstract Data and Communication Library (ADCL) is an empirical auto-tuning library which focuses on MPI communications.

1.6 Outline

An introduction to Finite Volume and spectral methods, the CFD applications and the HPC systems utilized in this work is given in chap. 2. Chapter 3 presents an overview over auto-tuning software, introduces ADCL and explains the challenges faced in this work. The extension of ADCL to collective communications is explained in chap. 4. Chapter 5 and 6 concentrate on the fundamental issues when using auto-tuning frameworks to tune MPI communications: outlier handling and the generation of empirical data. A summary and outlook in chap. 7 conclude this work. The glossary in the appendix gives an overview over specific terms used in this work.

2 Fundamentals

Computational Fluid Dynamics or CFD describes the study of fluid flows using computers. It involves solving the governing equations of fluid dynamics numerically on a geometrical domain divided into small volumes, i.e. on a mesh or grid. The processes of discretization refers to the transformation of the partial differential euqations (PDEs) into equivalent algebraic relations.

The application areas of CFD range from weather and climate simulations, over optimization of water or combustion power plants, circulations around cars, air foils and space crafts, analysis of pollution to blood flow simulations before stent insertions only to name a few. Combined with the magnetic effects modeled by Maxwell's equations, they can be used in magneto-hydrodynamics to study the dynamics of electrically conducting fluids such as plasma.

CFD has become a vital part in research and industry as it offers a number of advantages over traditional experiments:

- it allows to study fluid flows in locations that are not accessible to measuring heads, e.g. inside a furnice or an artery
- the fluid flow field can be observed without disturbing the flow itself
- CFD imposes less time constraints compared to limited time at experimental sites such as wind-tunnels
- CFD is of great economical value as simulations are much cheaper than experiments. In the design phase or when improving existing products, CFD can be used for parameter studies to reduce the number of prototypes and thus lower development costs significantly.

Despite the numerical advances in CFD, the solution of many flow phenomena is still far beyond present capabilities of workstations or small clusters and requires HPC. This is due to the shear size of the problems or/and the desire to model the physics as close to reality as possible. Time dependence, turbulence and the study of coupled phenomena (aero-acoustics, fluid-structure-interaction, multi-phase flows) add to the demands of computational power.

After introducing the governing equations of fluid dynamics in sec. 2.1, two computational methods, the finite volume method (FVM) and the spectral method will be presented in sections 2.2 and 2.3. They represent the two dominating

communication patterns, neighborhood communications and collectives, which are present in practically every MPI parallel application. Thus, results obtained in this thesis e.g. for the FVM would also extend to parallel solvers or a finite element discretization on a structured grid, as the underlying communication pattern is the same. Section 2.4 presents the application codes and test cases and sec. 2.5 the HPC systems used in this thesis.

2.1 The governing equations

The motion of fluids can be described in their most general form by the Navier-Stokes equations (NSEs), named after Claude-Louis Navier and George Gabriel Stokes. The compressible NSEs are based on conservation laws for mass, moment and energy with friction and heat transfer. They form a system of non-linear hyperbolic-parabolic equations. Depending on the type of PDE, the equations behave differently, both physically and numerically.

Under certain circumstances, simplifications are possible. If the Mach number is small, $M = \frac{|v|}{c} \rightarrow 0$ in theory and $M < 0.1 - 0.3$ in practice, the equations become incompressible and form a system of parabolic-elliptic PDEs. The incompressible Navier-Stokes equations on a domain Ω are usually written as

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\nabla P + \nu \mathbf{\Delta u}, \tag{2.1}$$

$$\nabla \cdot \mathbf{u} = 0, \tag{2.2}$$

where $\mathbf{u}$ is the velocity vector, P the pressure, ν the kinematic viscosity and Δ the Laplacian. Depending on the type of PDE, appropriate initial and boundary conditions need to be specified.

If heat transfer as well as friction ($Re \rightarrow \infty$) can be neglected, the NSEs simplify to the hyperbolic equations of gas dynamics, the Euler equations. The Reynolds number $Re = \frac{\rho v L}{\eta}$, where ρ is the density of the fluid, v the mean fluid velocity, L the characteristic linear dimension and η the dynamic viscosity of the fluid, is an important characteristic from which one can deduce the interaction of friction and velocity.

In differential form, the Euler equations are given as

$$\rho_t + \nabla \cdot (\rho \mathbf{u}) = 0 \tag{2.3}$$

$$(\rho \mathbf{u})_t + \nabla \cdot ((\rho \mathbf{u}) \otimes \mathbf{u}) + \nabla P = 0 \tag{2.4}$$

$$E_t + \nabla \cdot (\mathbf{u}\,(E + P)) = 0 \tag{2.5}$$

where ρ is the fluid mass density, $\mathbf{u}$ the fluid velocity vector, $E = \rho e + \frac{1}{2}\rho \mathbf{u} \cdot \mathbf{u}$ the total energy, e the internal energy and P the pressure. Since this results in 3D in five equations with 6 unknowns, the equations are closed by the equation of state, e.g. with the ideal gas law

$$P = \rho\,(\gamma - 1)\,e \Leftrightarrow P = (\gamma - 1)\left(e - \frac{1}{2}\rho v^2\right).$$

Equation (2.4) can be rewritten in vector form as

$$\mathbf{u}_t + \mathbf{f}_1(\mathbf{u})_x + \mathbf{f}_2(\mathbf{u})_y + \mathbf{f}_3(\mathbf{u})_z = 0$$

with

$$\begin{array}{rcllllll}
\mathbf{u} & = & (\rho, & \rho u_1, & \rho u_2, & \rho u_3, & e &)^T, \\
\mathbf{f}_1(\mathbf{u}) & = & (\rho u_1, & \rho u_1^2 + P, & \rho u_1 u_2, & \rho u_1 u_3, & u_1(e+P) &)^T, \\
\mathbf{f}_2(\mathbf{u}) & = & (\rho u_2, & \rho u_1 u_2, & \rho u_2^2 + P, & \rho u_2 u_3, & u_2(e+P) &)^T \quad \text{and} \\
\mathbf{f}_3(\mathbf{u}) & = & (\rho u_3, & \rho u_1 u_3, & \rho u_2 u_3, & \rho u_3^2 + P, & u_3(e+P) &)^T,
\end{array}$$

where the vectors $\mathbf{f}_i$ are fluxes.

2.2 Finite Volume Methods

A very popular numerical discretization method used in CFD is the finite volume method (FVM) where the PDEs are converted into a numerical scheme using a physically-based approach based on conservation laws. The domain is divided into small control volumes, also known as cells, elements or finite volumes, where the state variables are located at the cell center. Then, the differential form of the governing equations is integrated over a control volume. The resulting integral formulation is evaluated using Gauss integration, entailing the calculation of fluxes across cell boundaries. A suitable time discretization completes the numerical scheme.

The major advantage of the FVM is that for any single or set of control volumes the resulting solution exactly satisfies the conservation of mass, momentum and energy.

2.2.1 Discretization in space

The differential form of the conservation law

$$\mathbf{u}_t + \nabla \cdot \mathbf{f}(\mathbf{u}) = 0 \tag{2.6}$$

where $\mathbf{u}$ is a conserved quantity and $\mathbf{f}(\mathbf{u}) = [\mathbf{f}_1(\mathbf{u}), \ldots, \mathbf{f}_d(\mathbf{u})]$ is the flux vector, is integrated over a finite volume $V_i \in \mathbb{R}^d$ resulting in

$$\int_{V_i} \mathbf{u}_t dV + \int_{V_i} \nabla \cdot \mathbf{f}(\mathbf{u}) dV = 0. \tag{2.7}$$

Using the divergence theorem, one obtains

$$\int_{V_i} \mathbf{u}_t dV + \oint_{\partial V_i} \mathbf{f}(\mathbf{u}) \cdot \mathbf{n} dS = 0.$$

The vector $\mathbf{n} \in \mathbb{R}^d$ is the unit outward normal to the surface ∂V_i of V_i. To obtain a numerical scheme, one considers mean values over the volumes V_i, i.e.

$$\mathbf{u}_i = \frac{1}{|V_i|} \int_{V_i} \mathbf{u} \, dV$$

leading to

$$|V_i| \frac{\partial \mathbf{u}}{\partial t} + \oint_{\partial V_i} \mathbf{f}(\mathbf{u}) \cdot \mathbf{n} dS = 0. \tag{2.8}$$

We define the residuum

$$R = -\frac{1}{|V|} \oint_{\partial V} \mathbf{f}(\mathbf{u}) \cdot \mathbf{n} dS \tag{2.9}$$

as the change of the integral mean value of the control volume. The goal of the finite volume method is to calculate in each iteration the residuum R.

The computational domain V is divided into non-overlapping finite volumes

$$V_i, i = 1, \ldots, N,$$

which represent the control volumina such that $V = \cup V_i$. To simplify things for the numerical scheme, V_i is assumed to have piecewise smooth edges (2D) or surfaces (3D), i.e. V_i is a polygon in 2D or polyhedra in 3D, eventually with curved surfaces.

The rest of the FV method is now outlined in 2D for the sake of simplicity. The edges resp. surfaces between two finite volumes V_i and V_j are denoted with e_{ij}.

Using the simplified geometry, R can now be rewritten as

$$R_i = -\frac{1}{|V_i|} \sum_{e_{ij} \subset \partial V_i} \oint_{e_{ij}} \mathbf{f}(\mathbf{u}) \cdot \mathbf{n} dS.$$

The integration is done with Gauss quadrature rules, for schemes of first and second order with one Gauss point at the center of the edge or face, resulting in

$$R_i = -\frac{1}{|V_i|} \sum_{e_{ij} \subset \partial V_i} |e_{ij}| g(\mathbf{u}_i, \mathbf{u}_j; \mathbf{n}_{ij})$$

g is a numerical flux function, i.e. a suitable approximation of $\mathbf{f}$.

Since the Euler equations are rotationally invariant, the integrand $\mathbf{f}(\mathbf{u}) \cdot \mathbf{n}$ can be written as

$$\mathbf{f}(\mathbf{u}) \cdot \mathbf{n} = T^{-1}\mathbf{f}(T\mathbf{u}),$$

where T is a rotational matrix.

The residual is

$$R_i = -\frac{1}{|V_i|} \sum_{e_{ij} \subset \partial V_i} |e_{ij}| T^{-1} g(T\mathbf{u}_i, T\mathbf{u}_j; (1,0,0)^T) \tag{2.10}$$

For the flux calculation, at each cell boundary, the state vector of the two adjacent cells is turned into a local coordinate system, the resulting 1D Riemann problem at the edge is solved and the computed flux is rotated back into the global coordinate systems to calculate the residual of the cell.

To solve the Riemann problem at the cell boundaries to obtain the numerical flux g, the Roe scheme [27] is used in this thesis as an approximate Riemann solver. The Roe flux is the solution of the exact Riemann problem of the linearized conservation equation

$$\mathbf{u}_t + A_{lr}\mathbf{u}_x = 0, \quad \mathbf{u}(x,0) = \begin{cases} \mathbf{u}_l & \text{if x} < 0 \\ \mathbf{u}_r & \text{if x} > 0 \end{cases} \tag{2.11}$$

The Roe matrix $A_{lr} = A(\bar{\mathbf{u}})$ with the mean values

$$\bar{u}_i = \frac{\sqrt{\rho_r}u_{ir} + \sqrt{\rho_l}u_{il}}{\sqrt{\rho_r} + \sqrt{\rho_l}}, i = 1, \dots, 3, \qquad \bar{H} = \frac{\sqrt{\rho_r}H_r + \sqrt{\rho_l}H_l}{\sqrt{\rho_r} + \sqrt{\rho_l}},$$

$$\bar{c}^2 = (\gamma - 1)\left(\bar{H} - \frac{1}{2}\bar{u}^2\right), \qquad H = \frac{e + P}{\rho}$$

is a valid solution of (2.11). Its eigenvalues are

$$a_1 = \bar{v}_1 - \bar{c}, \qquad a_2 = a_3 = a_4 = \bar{v}_1, \qquad a_5 = \bar{v}_1 + \bar{c},$$

and the eigenvectors are

$$\begin{aligned}
r_1 &= (1, & \bar{u}_1 - \bar{c}, & \quad \bar{u}_2, & \bar{u}_3, & \quad \bar{H} - \bar{u}_1\bar{c} &)^T, \\
r_2 &= (1, & \bar{u}_1, & \quad \bar{u}_2, & \bar{u}_3, & \quad \tfrac{1}{2}(\bar{u}_1^2 + \bar{u}_2^2 + \bar{u}_3^2) &)^T, \\
r_3 &= (0, & 0, & \quad 1, & 0, & \quad \bar{u}_2 &)^T, \\
r_4 &= (0, & 0, & \quad 0, & 1, & \quad \bar{u}_3 &)^T, \\
r_5 &= (1, & \bar{u}_1 + \bar{c}, & \quad \bar{u}_2, & \bar{u}_3, & \quad \bar{H} + \bar{u}_1\bar{c} &)^T.
\end{aligned}$$

With the notations

$$\begin{aligned}
\Delta\rho &= \rho_r - \rho_l, \\
\Delta m_i &= \rho_r u_{ir} - \rho_l u_{il}, i = 1, \dots, 3, \\
\Delta e &= e_r - e_l \text{ and} \\
\overline{\Delta e} &= \Delta e - (\Delta m_2 - \bar{v}_2 \Delta\rho)\bar{v}_2,
\end{aligned}$$

one obtains from the condition $\mathbf{u}_r - \mathbf{u}_l = \sum_{i=1}^{5} \gamma_i r_i$ the coefficients

$$\begin{aligned}
\alpha_2 &= \frac{\gamma - 1}{\bar{c}^2}\left[\Delta\rho\left(\bar{H} - \bar{u}_1^2\right) - \overline{\Delta e} + \bar{u}_1 \Delta m_1\right] \\
\alpha_1 &= -\frac{1}{2\bar{c}}\left[\Delta\rho(\bar{u}_1 + \bar{c}) - \Delta m_1\right] - \frac{1}{2}\alpha_2 \\
\alpha_3 &= \Delta m_2 - \bar{u}_2 \Delta\rho \\
\alpha_4 &= \Delta m_3 - \bar{u}_3 \Delta\rho \\
\alpha_5 &= \Delta\rho - \alpha_1 - \alpha_2
\end{aligned}$$

Hereby one obtains the Roe flux

$$g_{\text{Roe}}(\mathbf{u}_L, \mathbf{u}_R) = \tfrac{1}{2}\left(\mathbf{f}_1\left(\mathbf{u}_L\right) + \mathbf{f}_1\left(u_R\right)\right) - \frac{1}{2}\sum_{k=1}^{5} \alpha_k |a_k| r_k$$

2.2.2 Discretization in time

Integrating (2.8) over a finite time interval $[t^n, t^{n+1}]$ and taking into the definition of the residuum (2.9) leads to

$$\int_{t^n}^{t^{n+1}} R_i dt = \int_{t^n}^{t^{n+1}} (\mathbf{u}_i)_t dt = \mathbf{u}_i^{n+1} - \mathbf{u}_i^n.$$

The time integral of R can be computed for a first order scheme with the midpoint rule, i.e.

$$\int_{t^n}^{t^{n+1}} R_i dt = (t^{n+1} - t^n) R_i(t^n).$$

This results in the fully discretized numerical scheme

$$\mathbf{u}_i^{n+1} = \mathbf{u}_i^n + \Delta t R_i^n = \mathbf{u}_i^n - \frac{\Delta t}{|V_i|} \sum_{e_{ij} \subset \partial V_i} |e_{ij}| T^{-1} g(T\mathbf{u}_i^n, T\mathbf{u}_j^n; (1,0,0)^T)$$

Explicit time discretizations are only conditionally stable, i.e. the maximum time step is limited, since we assume that the flux over a cell boundary is constant for

the time step. In 1D, this can be described descriptively: it is only the case if no waves from local Riemann problems over cell boundaries meet a neighboring cell boundary. This results in a local time step

$$\Delta t_i = CFL \cdot \frac{\Delta x_i}{|v_{1i}| + c_i}$$

for each cell i, where CFL is the CFL number, named after Courant, Friedrichs and Levy. Since the states in the cells are normally not the same, the global time step

$$\Delta t^{\text{global}} = CFL \cdot \min \frac{\Delta x_i}{|v_{1i}| + c_i}$$

is used. For multiple dimensions, the equation is modified into

$$\Delta t^{\text{global}} = CFL \cdot \min \frac{2r_i}{|v_i^{\max}| + c_i},$$

where r_i is the incircle diameter.

2.2.3 Boundary conditions

The boundary conditions are normally implemented with ghost cells. Wall, periodic, inflow or outflow boundary conditions are normally employed.

2.2.4 Higher-order schemes

Because of the first order discretization in space, only integral mean values exist in each cell. The values at the cell boundaries, which are the main input to the Riemann solver, differ greatly from the real ones. Using reconstruction, local values are computed from the integral mean values at the cell boundaries. This requires the knowledge of the distribution of the state variables inside a cell. For a second-order scheme, a piecewise linear distribution is assumed in the cell. The integral mean value of the cell has to remain unchanged. The necessary gradients are reconstructed with help of the neighboring cells. It is important, that the Total Variation Diminishing (TVD) property holds and no new minima and maxima are created. The reconstructed slopes thus have to be limited, in this case using Sweby's limiter with $\Phi = 1.5$:

$$s_\Phi(a,b) = sign(a) \max \{|\text{minmod}(a, \Phi b)|, |\text{minmod}(\Phi a, b)|\} \text{ with } 1 \leq \Phi \leq 2$$

where a and b are two scalar gradients, where minmod is defined as

$$\text{minmod}(a,b) = \begin{cases} a & \text{if } |a| < |b|, ab > 0 \\ b & \text{if } |a| \geq |b|, ab > 0 \\ 0 & \text{else} \end{cases} .$$

For Cartesian grids, the 1D scheme can be applied successively in each dimension whereas for unstructured grids, the different directions can no longer be treated separately and more complicated limiters are necessary.

A higher order discretization in time can be achieved with two different approaches: using separate discretizations for time and space, known as method of lines, or a combined space-time-discretization, the so-called space-time-expansion. The latter will be discussed in detail.

Equation (2.7) can be written as numerical scheme

$$\mathbf{u}_i^{n+1} = \mathbf{u}_i^n - \frac{1}{V}\int_{t^n}^{t^{n+1}} \sum_{m=1}^{N} g\left(\mathbf{u}_L^{(m)}(t), \mathbf{u}_R^{(m)}(t)\right) \mathbf{n} A_m dt$$

Instead of a quadrature rule for a fist-order scheme, we apply the an integration scheme of second order, e.g. mid-point rule

$$\int_{t^n}^{t^{n+1}} \sum_{m=1}^{N} g\left(\mathbf{u}_L^{(m)}(t), \mathbf{u}_R^{(m)}(t)\right) \mathbf{n} A_m dt = \Delta t \sum_{m=1}^{N} g\left(\mathbf{u}_L^{(m)}(t^{n+1/2}), \mathbf{u}_R^{(m)}(t^{n+1/2})\right) \mathbf{n} A_m$$

To calculate $\mathbf{u}_{L,R}^{(m)}(t^{n+1/2})$, we apply a Taylor expansion around the known state $\mathbf{u}_{L,R}^{(m)}(t^n)$,

$$\mathbf{u}(t^{n+1/2}) = \mathbf{u}(t^n) + \frac{\Delta t}{2}\mathbf{u}_t(t^n) + O(\Delta t^2).$$

To calculate the unknowns, the time derivate has to be determined. Because we focus on the state within one single cell and can presume that the distribution is continuous, eqn. (2.6) can be utilized. The Cauchy-Kovalevskaya (CK) procedure consists in solving it for the time derivative

$$\mathbf{u}_t = -\nabla \cdot \mathbf{f}(\mathbf{u}).$$

The complete idea of space-time-expansion is to carry out the Taylor expansion in time and space.

$$\begin{aligned} \mathbf{u}(x,y,z,t^{n+1/2}) \quad = \quad & \mathbf{u}(x_0,y_0,z_0,t^n) + \\ & \Delta x \mathbf{u}_x(x_0,y_0,z_0,t^n) + \\ & \Delta y \mathbf{u}_y(x_0,y_0,z_0,t^n) + \\ & \Delta z \mathbf{u}_z(x_0,y_0,z_0,t^n) + \\ & \tfrac{\Delta t}{2}\mathbf{u}_t(x_0,y_0,z_0,t^n) + O(\Delta x^2, \Delta y^2, \Delta z^2, \Delta t^2) \end{aligned}$$

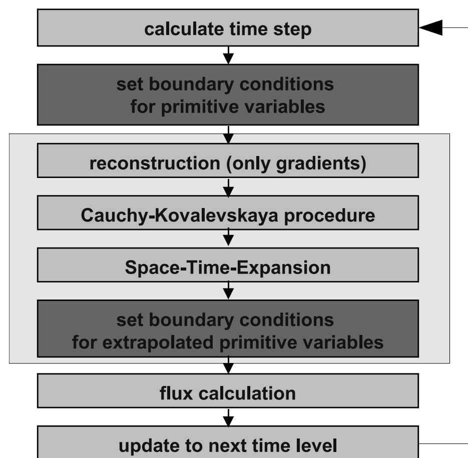


Figure 2.1: Schematic view of a second-order STE algorithm. Parts with data exchange are marked highlighted in dark grey.

The space derivatives are determined via reconstruction while the time derivatives are computed via the Cauchy-Kovalevskaya procedure.This makes it possible to determine the state at each point within a space-time-element. These states are used for the flux calculation.

Space-time expansion has some disadvantages over the method of lines. It is numerically less stable than e.g. Runge-Kutta procedures, and smaller time steps are necessary. Furthermore, the CK procedure is difficult to program. These disadvantages are more than compensated by the fact that reconstruction and flux calculation have to be executed only once instead of multiple times leading to shorter runtimes. The required synchronizations during each step of a Runge-Kutta method are omitted. In addition, the order in time and space do not have to be chosen seperately as a high order in space automatically yields a high order in time. This improves the quality of the solution compared to the method of lines.

2.2.5 Algorithm and Parallelization

Figure 2.1 shows the algorithm of a second-order STE algorithm.

A partitioning approach is used for parallelization. Each sub-domain is surrounded by one layer of ghost cells for the mean cell values as well as for the extrapolated variables. The data exchange happens when the boundary conditions for the cell mean values and for the extrapolated variables are set to provide the necessary values to compute the gradients and to apply the limiter. For a Cartesian grid, the resulting communication pattern, the neighborhood

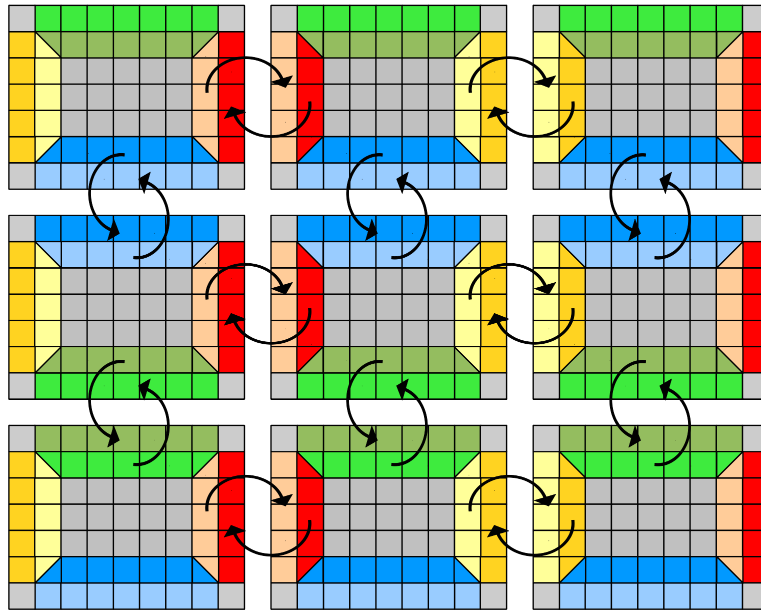


Figure 2.2: Neighborhood communication pattern to exchange layers of ghost cells for a 2D Cartesian grid.

communication, is shown in fig. 2.2. Each process exchanges data with its neighboring 4 (in 2D) and 6 (in 3D) processes.

2.3 Spectral Methods

Spectral methods belong to the group of traditional CFD methods that discretize and solve the NSEs directly to obtain hydrodynamic variables like density or the fluid velocity. Their properties make them well-suited for turbulence simulations. As the name suggests, spectral methods solve the NSEs with a spectral ansatz. The key part in the numerical solution of the resulting system of ordinary differential equations are Fast Fourier Transforms. This kernel has been integrated into several benchmarks, among them the NAS Parallel Benchmarks. This implementation constitutes one of the CFD applications of this work with which calculations on miscellaneous HPC systems were conducted.

Spectral methods [28, 29, 30, 31] have been developed since 1969. They use global polynomial or trigonometric polynomial basis functions of high degree which are mostly non-zero in the entire domain.

One of their main advantages is its exponential convergence, which means that the error is decreasing faster than any finite power of the number of grid points N. This results from the fact that an increase of N is equivalent to an increase

of the polynomial degree while at the same time $h = O(1/N)$ decreases. It guarantees an accurate solution of Poisson's equation for the pressure, which is responsible for mass conservation. In addition, it ensures a high resolution to obtain asymptotically correct higher-order statistics for small eddies[1] in turbulence studies at high Reynolds numbers which is computationally more cost-efficient than finite difference (FD) schemes. Another advantage of spectral methods is the low demand of main memory as spectral methods allow to obtain the same error with about half as many degrees of freedom than FD or finite element methods at higher cost per degree of freedom. This efficiency in memory consumption made spectral methods especially popular in weather simulation.

On the other hand, spectral methods generate full matrices, are harder to implement than FD schemes and affected more seriously in performance and accuracy from irregular domains compared to low-order alternatives.

2.3.1 Derivation of the spectral equations

Fourier spectral methods express the solution in space as a Fourier series

$$\begin{aligned} \mathbf{u}(\mathbf{x},t) &= \sum_k \hat{\mathbf{u}}_{\mathbf{k}}(t) e^{i\mathbf{k}\cdot\mathbf{x}} \\ p(\mathbf{x},t) &= \sum_k \hat{p}_{\mathbf{k}}(t) e^{i\mathbf{k}\cdot\mathbf{x}} \end{aligned}$$

and substitute this series into (a weak formulation of) the Navier-Stokes equations (2.1). Periodic boundary conditions in all three directions are assumed, where $\mathbf{u}$ is 2π-periodic with respect to $\mathbf{x}$. The initial conditions are $\mathbf{u}(\mathbf{x},0) = \mathbf{u}_0(\mathbf{x})$ in Ω at $t = 0$.

In Fourier space (2.1) and (2.2) become a system of ordinary differential equations (ODEs)

$$\left(\frac{\mathrm{d}}{\mathrm{d}t} + \nu|\mathbf{k}|^2\right)\hat{\mathbf{u}}_{\mathbf{k}} = -i\mathbf{k}\hat{p}_{\mathbf{k}} + \hat{\mathbf{c}}_{\mathbf{k}}, \tag{2.12}$$

$$i\mathbf{k}\cdot\hat{\mathbf{u}}_k = 0, \tag{2.13}$$

which contain the time-dependent coefficients of the trigonometric terms in the series written in complex exponential form.

The term

$$\hat{\mathbf{c}}_{\mathbf{k}} = -(\widehat{\mathbf{u}\cdot\nabla\mathbf{u}})_{\mathbf{k}}$$

[1] An eddy is the swirling of a fluid and the induced reverse current when the fluid flows past an obstacle.

is nonlinear and responsible for most of the algorithmic complexities of the problem. The pressure may be eliminated be taking $i\mathbf{k}$ dotted with (2.12) and using (2.13). Hence,

$$rcl\hat{p}_{\mathbf{k}} = -\frac{1}{|\mathbf{k}|^2} i\mathbf{k}\cdot\hat{\mathbf{c}}_{\mathbf{k}}$$

$$\left(\frac{\mathrm{d}}{\mathrm{d}t} + \nu|\mathbf{k}|^2\right)\hat{\mathbf{u}}_{\mathbf{k}} = \hat{\mathbf{c}}_{\mathbf{k}} - \mathbf{k}\frac{\mathbf{k}\cdot\hat{\mathbf{c}}_{\mathbf{k}}}{|\mathbf{k}|^2}.$$

The Fourier Galerkin approximation consists of truncating the sums at

$$|k_1|, |k_2|, |k_3| < N/2.$$

The system of ODEs is then solved by a time-stepping method.

In component form the nonlinear term is

$$(\hat{\mathbf{c}}_{\mathbf{k}})_\alpha = -i\mathbf{k}_\beta \sum_{\mathbf{m}+\mathbf{n}=\mathbf{k}} \hat{u}_{\beta,\mathbf{m}}\hat{v}_{\alpha,\mathbf{n}}. \tag{2.14}$$

This convolution sum in wave vector space is the standard Galerkin approximation to the nonlinear term $-\mathbf{u}\cdot\nabla\mathbf{u}$. A summand of (2.14) is a triple convolution sum

$$\hat{s}_{\mathbf{k}} = \sum_{\mathbf{m}+\mathbf{n}=\mathbf{k}} \hat{u}_{\mathbf{m}}\hat{v}_{\mathbf{n}}, \tag{2.15}$$

In one dimension, (2.15) simplifies to

$$\hat{s}_k = \sum_{m+n=k\pm N} \hat{u}_m\hat{v}_n$$

The pseudo-spectral transform method consists of transforming $\hat{u}_m$ and $\hat{v}_n$, $n = -N/2,\ldots,N/2-1$, to the physical space using the discrete Fourier transform (DFT) with N points, forming the physical space products $u_i v_i, i = 0,\ldots,N-1$, and then transforming these products back to Fourier space. The result is

$$\tilde{s}_k = \hat{s}_k + \sum_{m+n=k\pm N} \hat{u}_m\hat{v}_n.$$

2.3.2 Algorithm and Parallelization

A straightforward evaluation of the DFTs is tremendously expensive for large N. Steps towards efficiency include partial summations, i.e. using a sequence of nested one-dimensional transforms for multi-dimensional transforms, and an algorithm formulated by Cooley and Tukey [32] in 1965 to compute a DFT of n

points in $O(n\log n)$ steps which became known as Fast Fourier Transform (FFT). The computation of an three-dimensional (3D) FFT is the essential and most time consuming part of a DNS with a Fourier spectral method. It might account for more than 90% of the computing time as it involves a matrix transpose resulting in an collective communication of type `alltoall` which causes large amounts of data transfer [33, 34]. FFTs have been included in several benchmarks to evaluate network bandwidth. In this thesis, the 3D FFT kernel of the NAS parallel benchmarks is used.

2.4 Application Test Cases

This section gives a brief introduction into the two applications used in this thesis, the code Euler3D implementing a finite-volume method for the Euler equations and the NAS FT benchmark representing the kernel of a spectral method.

2.4.1 Euler3D

The code Euler3D [35] implements the finite volume method for the incompressible Euler equations in 3D on structured grids. It is an extension to the 2D variant used in courses and workshops at the Institute of Aero- and Gasdynamics of the University of Stuttgart to teach the basic principles of the FVM. Euler3D is written in Fortran90 using modules.

Its code structure follows the schematic view of fig. 2.1. As test case, we use a Gauss pulse which travels diagonally through a cubic domain as shown in fig. 2.3. Table 2.1 gives an overview of the problem sizes used and the resulting message sizes.

2.4.2 The NAS FT Benchmark

The NAS Parallel Benchmarks [36, 37, 38] from NASA Ames Research Center are "paper and pencil" benchmarks and consist of a set of five computational kernels (EP, MG, CG, FT, and IS) and three pseudo applications (LU, SP and BT). They emulate the characteristic features of large scale CFD applications and serve to evaluate supercomputers. Each of the kernels addresses a different type of numerical computation.

The FT benchmark from NAS Parallel Benchmarks 3.0 computes the solution of a 3D partial differential equation with a FFT which requires all-to-all communications for matrix transpose operations. Consequently, the FT benchmark

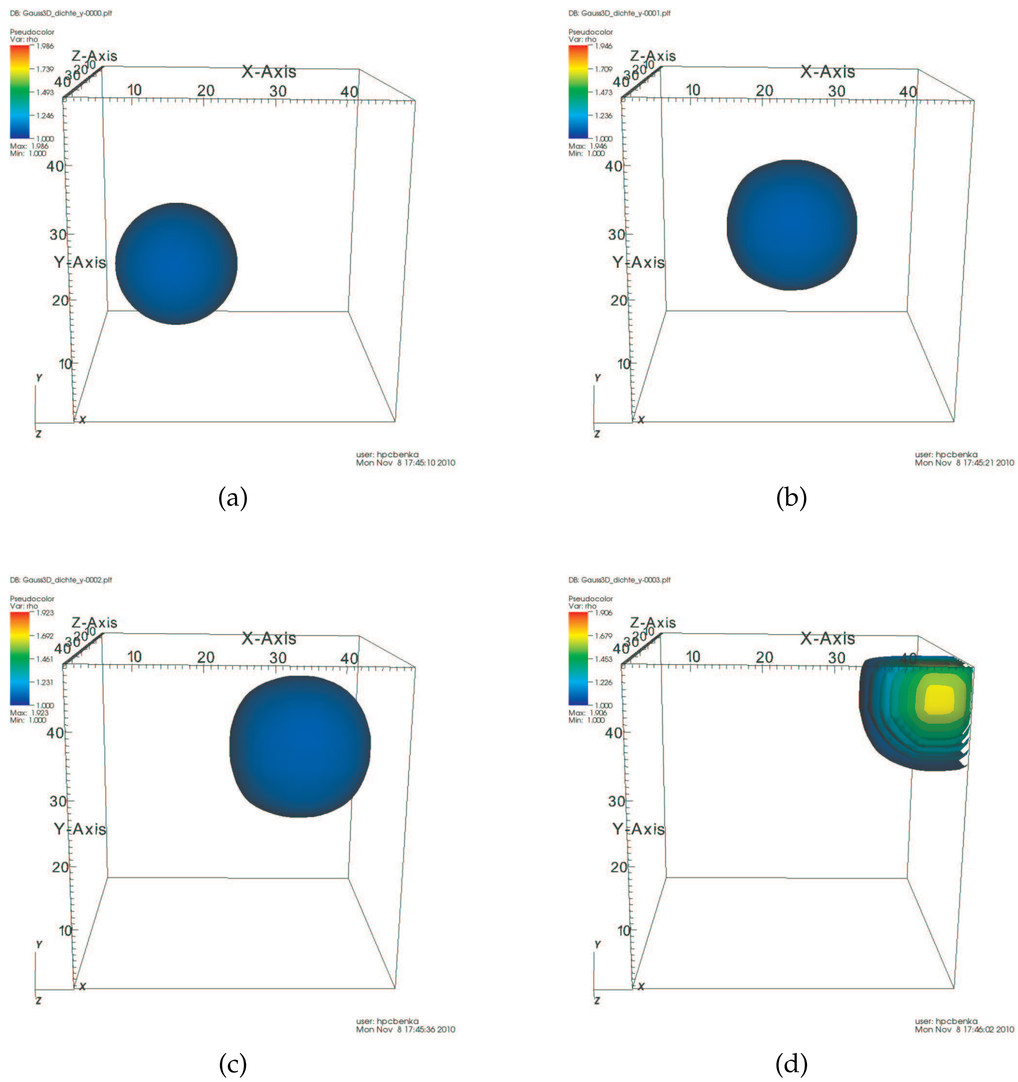


Figure 2.3: Test case: a Gauss pulse in density travelling diagonally through the computational domain.

#elements/process	class	#procs	message size per process [MB]	total message size[GB]
20^3		64 256 1024	0.7	0.04 0.17 0.69
30^3	XS	64 256 1024	1.5	0.10 0.39 1.55
40^3	S	64 256 1024	2.7	0.17 0.69 2.75
60^3	L	64 256 1024	6.0	0.39 1.55 6.19
80^3		64 256 1024	10.8	0.69 2.75 11.01

Table 2.1: Message sizes for Euler3D (weak scaling).

creates substantial communication and evaluates network performance. Its main loop consists of an evolution step and the computation of the FFT. The problem classes S (sample code) and A–E vary primarily in the size of the main arrays. A simulation of a class K benchmark running with n_p processes is referenced to by $n_p\,K$, thus $8A$ stands for a class A benchmark executed with 8 processes. The dimensions of the domain as well as the message sizes for the different test cases are shown in table 2.4.

Assume a grid with $n_1 \times n_2 \times n_3$ points. The 3D FFT is performed as a series of three successive 1D FFTs on one processor. Depending on the magnitude of n_p compared to n_3, each processors owns a number of contiguous x-y planes ($n_p < n_3$) or only a number of rows x ($n_p > n_3$). For the former, the 1D FFTs in the x and y directions are performed locally on one processor. Afterwards the data is transposed (using `MPI_Alltoall`) and the FFT in z-direction can be computed locally. For the latter, two transposes are performed to perform the FFTs in y- and z-direction. For our test cases, the number of processes always undermatches n_3 resulting in one collective per time step.

class	n_1	n_2	n_3	#procs	message size per process [KB]	total message size[GB]
A	256	256	128	8	2097	0.1
B	512	256	256	8 32	8388 524	0.5
C	512	512	512	32 128 256 512	2097 131 32 8	2.1

Figure 2.4: Message sizes for the FFT benchmark (strong scaling).

2.5 HPC Systems

The following HPC systems are used in this work:

The hierarchical network is described by the *blocking factor* which specifies the ratio of accumulated communication bandwidth. Suppose 30 nodes are connected to a level 2 switch (spine switch) with DDR and there are 6 QDR links between the level 2 and level 1 switches. The signaling rate is 5 Gbit/s for DDR and 10 Gbit/s for QDR in each direction per connection. The blocking factor is then

$$6 \cdot 10 : 30 \cdot 5 = 1 : 2.5$$

Cacau Cluster: The *cacau* cluster at High Performance Computing Center of Stuttgart (HLRS) consisted of 200 dual processor 3.2 GHz Intel EM64T processors, connected by a 4xInfiniBand network interconnect and a secondary hierarchical Gigabit Ethernet network. A total of six 48-port switches were used to connect the nodes, each 48-port switch had four links to the upper level 24 port Gigabit Ethernet switch. Thus, this network had a 12:1 blocking factor. In the mean time, the cluster is out of service.

Cray XT5m: The installation at HLRS has 112 Dual Socket Quad Core nodes consisting of AMD Opteron Processors 23 (C2)@2.4 GHz and 16 GB memory/node. Each SeaStar chip provides four network links connecting to four neighbors in the 2D torus, the peak bidirectional bandwidth of each link is 9.6 GB/s with sustained bandwidth in excess of 6 GB/s and 38.4 GB/sec switching capacity per chip.

IBM Blue Gene/P: *Jugene* at the Jülich Supercomputing Center is a 73728 node installation of 4-way Symmetric Multi-processing (SMP) 32-bit PowerPC 450 cores at 850 MHz with 2 GB memory each. The compute nodes are organized in 72 racks with 32 node cards each with 32 compute nodes.

The compute nodes are connected in a 3D torus with 5.1 GB/s per node and an additional collective network.

NEC SX-8: NEC SX-8 is a vector computer. Each shared memory node disposes of 8 CPUs running at 2 GHz, each with 16 GFlop/s peak performance and 16 GB of main memory. The nodes are connected with an IXS Crossbar with 8 GB/s bidirectional bandwidth. The number of nodes at HLRS has been reduced from 72 in 2004 to currently 10 nodes.

Laki Cluster: The NEC Nehalem Cluster *Laki* at HLRS consists of 700 dual socket quad-core compute nodes with Intel Xeon (X5560) Nehalem processors running at 2.8 GHz with 12 GB memory/node. The nodes are connected with InfiniBand DDR x4 (up to 20 GBit/s in both directions) with Voltaire Grid Director 4036 QDR switches. Up to 30 nodes are connected to a 36 port switch. The remaining six ports are used to connect to each of the six first level switches. The blocking factor is 1:2.5.

SGI Altix 4700: HLRB-II has 9728 cores organized in 19 compute partitions with 512 cores and 128 or 256 blades (memory channels) each. The processors are Intel Itanium 2 Montecito Dual Cores running at 1.6 GHz. Each core has 4 GB main memory. The NUMAlink4 interconnect has 2 (bidirectional) links per blade, with a total bandwidth of 12.8 GB/s.

Shark Cluster: The *shark* cluster at the University of Houston consists of 24 single processor, dual core 2.2 GHz AMD Opteron nodes. Each node is equipped with a 4xInfiniBand and a Gigabit Ethernet card. It disposes of an InfiniBand network and a Gigabit Ethernet network. The switch used within the Gigabit Ethernet network provides a full duplex 1 GBit connection for each node.

3 Automatic Performance Tuning

From its early beginnings in 1995 until now, automatic performance tuning has been applied to various application areas, among them linear algebra, FFTs, solvers and MPI communications. Its historic development is described in sec. 3.1. The underlying principles of automatic optimization systems, i.e. when and how to tune, are detailed in sec. 3.2. The Abstract Data and Communication Library is an auto-tuning library with special support to optimize MPI communications. Its functionality is presented in sec. 3.3. Section 3.4 discusses the necessary extensions which are realized in this work.

3.1 Historic Overview

The first automatic tuning frameworks focused on linear algebra and Fast Fourier Transforms. After efforts to formalize automatic tuning, its principles were employed in different application areas, most importantly in parallel linear algebra, for solvers and communication operations.

3.1.1 The Beginnings

Numerical linear algebra (i.e. solution of linear systems, linear least squares problems, eigenvalue problems and singular value problems) is a fundamental part of many scientific calculations. It in turn is based on small, computationally intensive low-level operations, namely vector and matrix operations. These operations were encapsulated in an API and became known as BLAS (Basic Linear Algebra Subroutines) [19, 20, 21, 39]. Driven by the need to offer high-performance, low-cost BLAS routines on a steadily increasing number of architectures, Bilmes et al. presented PHiPAC (Portable High-Performance matrix-vector libraries in ANSI C) [40] in November 1995. It included a parameterized code generator for the general matrix-matrix multiply (GEMM in BLAS) $C = \alpha op(A) op(B) + \beta C$ where $op(X) = X$ or X^T as well as search scripts for a brute force and a more sophisticated search technique including a model-based approach to identify best parameters. With the code generator, a large set of variants could be created at install time of PHiPAC which provided the library with

a rich choice of optimization options. Coding guidelines for high-performance ANSI C exposing a high degree of optimization possibilities to the compiler ensured portability. Experiments showed that 90% of peak performance on a variety of current workstations was obtained and vendor-supplied libraries were often out-performed. Although being the first automatic performance library, PHiPAC did not reach the popularity of its successor ATLAS.

In September 1997, Frigo introduced FFTW 1.0 [41], a portable C library for computing the complex discrete Fourier transform (DFT) in one or more dimensions. It consists of three major parts: a codelet generator which automatically produces code to solve various small sized DFTs, a planner that tests different strategies how to combine the codelets based on a divide-and-conquer approach and an executor that executes the plan. It proved to be faster compared to other DFT software (FFTPACK, code from Numerical Recipes) and vendor-supplied code. In contrast to PHiPAC, optimization is purely done at runtime, performance measurements are based on multiple executions of the same DFT and the search follows a dynamic programming algorithm. The author valued the creation of DFTs for larger radices not practical to implement manually and the ease to implement diverse optimizations and algorithmic variations. At the time of publication, an MPI parallelization was added. In the latest release FFTW3, the runtime structure was redesigned to allow for a much larger search space.

Like PHiPAC, the first version of Automatically Tuned Linear Algebra Subroutines (ATLAS) [42] released in December 1997 [43] focused on the GEMM and reports performance comparable or superior to vendor-tuned libraries. The code generation possibilities have been gradually extended to support GEMM-based Level 3 BLAS [44, 26], the full BLAS [26] as well as some higher level routines of the LAPACK API. Whereas PHiPAC uses a wholistic approach tuning each matrix operation separately, ATLAS assumes an on-chip cache, i.e. L1 cache, which is accessible from the floating point unit and breaks all operations down to a square on-chip matrix multiply. This results in shorter tuning times due to only one tuning operation and a small number of tuning parameters dealing with the properties of the L1 cache. The tradeoff is lower performance for small matrices and on systems without L1 cache. ATLAS runs a set of micro-benchmarks to identify hardware parameters (size of L1 cache, number of floating point registers, latency of floating-point multiplications, etc.) which are then used to bound the search space. In an orthogonal line search where each of the parameters is tuned separately one after the other, ATLAS discovers an approximate minimum. The order of the parameters, the set of possible values for each parameter and the reference values for not-yet optimized parameters influence the result of the search. With ATLAS Unleashed, hand-tuned implementations can be added to the search space.

In the domain of signal and image processing, an additional two projects apart

from FFTW surfaced around 2001, SPIRAL and UHFFT. SPIRAL [45, 46] is an ongoing project to optimize Fourier transforms. The transform algorithms are described in a symbolic Kronecker-product based language SPL and translated into enhanced C code with a special purpose compiler. SPIRAL employs search methods based on genetic algorithms [47, 48] at compile time over a space of mathematically equivalent formulas. UHFFT [49] is similar to FFTW when regarding the runtime search with a planner step. In addition, it provides a search method based on performance data gathered at install-time. Transforms are broken up into smaller parts and executed taking into account the users' specifications for FFT sizes.

3.1.2 Formalization efforts

At the same time, first approaches were taken to formalize automatic tuning.

The ATLAS project described its ideas as Automated Empirical Optimization of Software (AEOS) [26], a paradigm for high performance library production and maintenance. Whaley et al. defined that a portable performance-critical library using the AEOS methodology (1) requires to isolate performance-critical routines and has (2) methods of adapting software to differing environments, (3) robust, context-sensitive timers and (4) an appropriate search heuristic. The methods of software adaption are subdivided into *parameterized adaption* (characteristics which vary from machine to machine e.g. blocking factor are parameterized) and *source code adaption* for those architectural variables where parameterized adaption fails (e.g. instruction cache size, pipeline length). Source code adaption can be achieved via *multiple implementations* (various hand-tuned implementations enriched with a search and timing layer) and *code generation* based on parameters offering a high flexibility, high complexity, specialty-based approach. An AEOS provides many ways of doing a required operation and empirical timings in order to choose the best method for a given architecture. It automatizes the already to some extend empirical hand-tuning process.

A broader approach was taken for the Self-Adapting Numerical Software (SANS) concept [50]. It attributes tuning to different classes such as algorithms, networking as well as data layout or kernels. Each class is influenced by different factors and can be optimized before or only at runtime.

3.1.3 Linear algebra

LAPACK for Clusters (LFC) [51, 52, 53] offers automated tuning of parallel computing resources for LAPACK. It was initially presented at ParCo 2001 and later adapted to the context of SANS. It hides complexity from the non-expert

user by providing a serial, single processor user interface including resource discovery. LFC decides based on cluster state and overhead whether to execute the routine in serial or parallel. LFC's main components are a data collection daemon, a data mover, a performance modeler/resource selector and a great number of selection (annealing, ad-hoc evaluation, genetic algorithms, dynamic linear programming) and search methods (entire parameter space, generic black box optimization technique or reduce search space through domain-specific knowledge) with historic learning. Tuning happens at compilation time or dynamically during execution (at runtime) with or without monitoring.

Similar to LFC, DESOBLAS [54] also hides the choices of performance parameters from the user. The library implements a parallel version of BLAS based on the idea of delayed evaluation. After a sequence of calls to the library which immediately return, the user has to explicitly request the execution of the pending operations. DESOBLAS takes into account all outstanding operations to decide on the data layout.

3.1.4 Solvers

SPARSITY [55, 56] is a toolkit that automatically generates implementations for the sparse matrix-vector multiplication $y \leftarrow Ax + y$ with one or more vectors. It combines ideas from automatic performance tuning and the developing Sparse Basic Linear Algebra Subroutines (SpBLAS) [57]. Since SpBLAS hides the data structures from the user, an automatic tuning library can freely decide on data structure and implementation. Since register and cache blocking as well as an adequate choice for the block size highly depend on the matrix, the automatic system decides based on a sample matrix on when and how to optimize. Performance models based on machine profiling help to prune the search space. Code generation is partially automatized and produces C code. A comparison of optimized and unoptimized version ensures no performance loss.

The Optimized Sparse Kernel Interface (OSKI) is a collection of low-level primitives that provide automatically tuned computational kernels on sparse matrices and continues the work on SPARSITY. Apart from offering transparent access to a variety of performance optimization techniques, the costs of tuning are unfolded.

The software package AcCELS (Accelerated Compress-storage Elements for Linear Solvers) [58] optimizes automatically sparse-matrix-vector operations. It extends the work of Vuduc [56, 59]. At install time, parameters are estimated, measured and interpolated performance values are tabulated. At runtime, the sparse matrix is analyzed. An estimation of the fill-in and the optimal

performance of a matrix-vector product for a given block size is calculated based on three parameters.

The Self-Adapting Large-scale Solver Architecture (SALSA) [60] is a project aiming at providing the suitable linear and non-linear system solver to an application. Using the characteristics of the application matrix, the solver contacts a knowledge database and provides an estimate on the best solver to use. Among the characteristics used for choosing the right solver are structural properties of the matrix (e.g. maximum and minimum number of non-zeros per row), matrix norms such as the 1- or the Frobenius-norm, and spectral properties.

3.1.5 Automatic performance tuning for MPI communications

Some recent projects focus on automatic performance analysis of MPI communications. These are:

ACCT [61, 62] aimed at optimizing collective communications. It focuses on broadcast, scatter and gather since all other operations can be implemented with these three collectives. The library executes a series of experiments to determine the best buffer size and the best algorithm for various number of processes. To reduce the number of experiments, only certain points within the search space are evaluated and later on, a model-based approach was added. The library is equipped with two modified steepest descent algorithms to prune the search space at install time.

STAR-MPI [63], introduced in 2006, allows the automatic optimization of collective MPI operations (alltoall, allgather, allgatherv, allreduce, broadcast) at runtime providing a similar API as defined in the MPI specifications [18]. It comprises of an algorithm repository for each collective, a simple automatic selection mechanism as well as a monitoring interface. The algorithm repository consists of topology unaware algorithms and automatically generated topology aware ones. Both can be annotated with meta-data concerning relevant system parameters. For a faster tuning, "algorithm grouping" similar to ADCL attributes is introduced. The library initially compares a single algorithm from all available groups. After the winner group has been determined, the library does a fine-tuning of the performance by evaluating all other available algorithms within the winner group. During the monitoring phase, allreduces are used from time to time, and depending on the outcome, the interval is increased, stays the same or the algorithm is switched. To use the library, calls to collective MPI routines are prefixed with STAR_. Site-dependent and message-dependent optimization is supported.

In 2008, Hartmann et al. [64] presented a tuning framework for specific collectives. Its main idea is to use orthogonal processor layouts. A configuration phase explores possible performance improvements which is the basis for the decision on the implementation during the execution phase.

3.2 Characteristics of automatic tuning systems

Tuning systems vary in many ways, for example in how codelets, i.e. implementation alternatives, are provided (e.g. produced by a parameterized code generator in a high-level language to ensure portability and leverage advances in compiler technology automatically), how to prune the search space (exhaustively, with heuristics or model-based) and when the tuning is done (at install-, compile- or runtime or any combination of them). This section will look into the different possibilities of how and when to tune.

3.2.1 When to Tune

Automatic optimization libraries can either apply static tuning or dynamic tuning or a combination of both. *Static tuning* is the process of optimizing a code sequence/application a priori of the actual execution of the program resulting in software that cannot alter its behavior during execution. *Dynamic* or *runtime tuning* on the other hand leads to software that has the ability to adapt its behavior at runtime. Static tuning is sometimes subclassified into *install-time* tuning at the installation of the automatic optimization library and *compile-time tuning* when compiling the application.

The point in time to tune depends partly on the availability of all relevant information which shifts with the area of application. For example for dense matrix kernels, install- or compile-time tuning is quite sufficient [40, 26], whereas for tuning of MPI communications runtime information is crucial.

For MPI communications, several factors influencing the performance of the application can only be determined while executing the application. Faraj et al. acknowledge the fact, that application characteristics (e.g. communication volume depending on input data, communication frequency) and process placement by the batch scheduler significantly influence the performance [65]. The latter results in non-uniform network behavior [66]. Other factors are shared resources such as network links and switches or file systems, the influence of process arrival patterns to the performance of collective communication operations as well as OS jitter [67]. OS jitter leads to a slow-down of a subset of processes utilized in a parallel job. Depending on the work that each process has

to perform, the order in which processes start to execute a collective operation varies strongly depending on the application. Thus, the algorithm determined to lead to the best performance using a synthetic benchmark might in fact be non optimal in a real application [68].

Inherent to the tuning system is the introduced overhead. The losses for runtime tuning — mainly introduced by the search process — have to be amortized over frequent uses of the kernel while install- or compile-time tuning hide the extra time from the user. From a systems administrator's point of view, static tuning is resource-intensive and a waste of computation time. The tuning procedure itself often exceeds the runtime of an individual application and most of the pre-tuned sets of parameters are never used. They will typically not reserve the according time slots to tune these libraries exhaustively in advance, but only a limited time, e.g. some hours instead of multiple days to tune the MPI collective operations on a multi-thousand node cluster. End-users themselves will most probably not use their valuable compute time to perform these time consuming operations.

3.2.2 How to Tune

To determine the *optimal codelet*, i.e. the implementation which gives the best performance for the complete simulation, the performance of a set of codelets has to be assessed. This can be done in two ways or a combination of them. One way is heuristic modelling where parameterized theoretical models are used to estimate the execution times of the codelets. The other is empirical evaluation of the codelets, i.e. the codelets are run and their performance is measured.

3.2.2.1 Heuristic modelling

Heuristic modelling predicts the execution time of codelets and compares the predicted execution time of various algorithms. A *heuristic* is a strategy to find a close-to-optimal solution in a relatively short time. In the case of communication optimization, performance predictions are based on *theoretical* communication models such as Hockney's model [69], LogP [70], and LogGP [71]. This approach has been taken in [72].

Hockney's model applies to point-to-point operations. It models the communication time as

$$t(n) = t_0 + \frac{n}{r_\infty}$$

where n is the message length in bytes, t_0 is the latency and r_∞ the asymptotic bandwidth, i.e. the maximum achievable bandwidth when $n \to \infty$.

The LogP model estimates the communication time of fix-sized short messages using four parameters L, o, g and P. L is an upper bound on the latency and P the number of processors. The overhead o is the time span during which the processor is engaged into sending or receiving the message and its reciprocal corresponds the communication bandwidth. The gap g defines the minimum time between sending or receiving two consecutive messages, its reciprocal corresponds to the communication bandwidth.

The LogGP extends the LogP model with a linear model for long messages by adding one additional parameter, G, which describes the bandwidth obtained for long messages.

3.2.2.2 Empirical Evaluation

Another way is to empirically evaluate the codelets, i.e. to actually run them. This requires methods to prune the search space of available codelets. The *selection logic* chooses the order in which the codelets are evaluated. The *decision logic* produces a decision based on the collected empirical data and determines the optimal codelet.

In case of a non-existing selection logic, an *exhaustive* or *brute force search* will be conducted which tests all available codelets. This guarantees to find the best implementation, but requires considerable amounts of computing time. This method is therefore ill-suited for runtime optimizations with large numbers of codelets where the overhead of the expensive search phase probably does not pay off during the time slot accorded by the batch scheduler.

To limit computational effort, the search space can be bounded, e.g. until performance starts to decline, by a branch-and-bound technique or arbitrary restrictions such as looking only at block sizes which are powers of two. Several systems employ steepest descent methods although it is a discrete optimization problem and jumps in performance between algorithmic variants are possible as e.g. shown in [73]. Seymour et. al. evaluated the effectiveness of Nelder-Mead simplex, genetic algorithms, simulated annealing, particle swarm optimization, orthogonal search, and random search [74]. Depending on the case, the particle swarm optimization or the orthogonal search were advantageous.

Vuduc [73] considered the problem of search in an abstract framework, defining an early-stopping criterion and elaborating a way how to classify implementations depending on input data at runtime.

The effectiveness of search strategies ultimately depends on characteristics of the optimization space.

3.2.2.3 Advantages and Disadvantages of Heuristic Modelling and Empirical Evaluation

When regarding communication models, some of them are highly sophisticated, but ultimately suffer from three limitations: firstly, it is often hard to determine some parameters of (sophisticated) communication models. As an example, no approach is published as of today which derives a reasonable estimate of the receive-overhead in the LogGP model [75]. Second, while it is possible to develop a performance model for a simple MPI-level communication operation, more complex functions involving alternating and irregular sequences of computation and communication have hardly been modeled as of today. Lastly, all models have their fundamental limitations and break-down scenarios, since they represent simplifications of the real world behavior of the machines. Thus, while modeling collective communication operations can improve the understanding of performance characteristics for various algorithms, tuning complex operations based on these models is fundamentally limited. Fagg confirms in [61] that the "randomness of our results for a given system also show that a generalized mathematical model will often not be able to give optimal performance". Pješivac-Grbović [76] acknowledged that "based on our findings, we believe that the complete reliance on models would not yield optimal results."

To outperform empirical search methods, analytical models have to be accurate enough to be useful, yet simple enough to evaluate quickly. Faulty assumptions or imprecisions in the model which are likely to occur for purely static models can lead to a non optimal choice of parameter values. As Kulkarni put it in [77]: "one advantage of empirical search over analytical models [is]: the *lack* of intelligence of the search can occasionally provide better results when the platform behaves in a manner other than the model's expectation". This is why the empirical approach has mostly displaced theoretical modelling and is now widely accepted.

3.3 The Abstract Data and Communication Library (ADCL)

The Abstract Data and Communication Library (ADCL) [78] is an automatic empirical performance tuning framework at application level and was introduced in 2007 by Edgar Gabriel and Shuo Huang [79]. As any auto-tuning library, it aims to provide the highest possible performance within a given execution environment. Its use is advantageous as only a single version of the application source code needs to be maintained while still having the ability to achieve close

to optimal performance for the application on all architectures. Furthermore, ADCL gives application developers the possibility to express often occurring communications patterns in higher level terms and takes away complexity from them while at the same time exploiting optimization possibilities. The main area of application are iterative procedures with repetitive numerical kernels or communication patterns which may stem from iterative solvers or time discretization schemes. The library is written in C and provides a Fortran interface. A key concept of the adaptive communication framework is its ability to select the fastest of the available codelets during the regular execution of the application. This allows ADCL to adapt itself to the characteristics of the current hard- and software environment as well as the application and optimize also parameters which are only known at runtime. ADCL uses the first executions of the communication within the application to determine the fastest available codelet. Although some of the tested codelets will deliver a non optimal performance, this approach avoids a separate 'planner' step or expensive pre-tuning which does not contribute to the simulation. This approach also allows an easy integration of a monitoring interface when tuning MPI communications which could restart the runtime selection logic in case the networking conditions changed significantly compared to the initial evaluation.

ADCL incorporates a runtime selection and decision logic in order to choose the codelet leading to the highest performance of the application. Two different runtime selection algorithms are currently available within ADCL: the library can either apply a brute force search strategy which tests all available codelets of a given communication pattern; alternatively, a heuristic relying on attributes characterizing a codelet has been developed in order to speed up the runtime selection procedure [79].

The primary purpose of the library is as an add-on to MPI to optimize MPI communications and therefore ADCL requires its own API which will be introduced in sec. 3.3.1. However, in principle, the functionality of ADCL could also be used to optimize other applications. The user has to implement different codelets for a particular problem, register them with the library and define an user-provided set of codelets. Then, he can employ the selection and decision algorithms of ADCL to determine the fastest way to solve the problem.

In the past, ADCL has been used to tune parallel matrix-matrix operations, low-level parameters of a message passing library [80] and the n-dimensional neighborhood communication, where processes are located in a Cartesian grid and each process communicates with its next neighbors along each of the n axes. This highly relevant communication pattern is the dominant communication operation in many applications [81, 82] with stencil computations. At present, the library provides a large predefined set of 20 codelets for this pattern shown in tbl. 3.1. The different codelets of the neighborhood communication provided

no	Codelet name	Comm. pattern	Handl. of non-cont. msgs.	Data transfer primitive
0	IsIr_aao	all-to-all	derived types	`MPI_Isend/Irecv/Waitall`
1	IsIr_pair	pair-wise	derived types	`MPI_Isend/Irecv/Wait`
2	IsIr_aao_pack	all-to-all	pack-unpack	`MPI_Isend/Irecv/Waitall`
3	IsIr_pair_pack	pair-wise	pack-unpack	`MPI_Isend/Irecv/Wait`
4	SIr_aao	all-to-all	derived types	`MPI_Send/Irecv/Waitall`
5	SIr_pair	pair-wise	derived types	`MPI_Send/Irecv/Wait`
6	SIr_aao_pack	all-to-all	pack-unpack	`MPI_Send/Irecv/Waitall`
7	SIr_pair_pack	pair-wise	pack-unpack	`MPI_Send/Irecv/Wait`
8	S_R_pair	pair-wise	derived types	`MPI_Send/Recv`
9	Sr_pair	pair-wise	derived types	`MPI_Sendrecv`
10	S_R_pair_pack	pair-wise	pack-unpack	`MPI_Send/Recv`
11	Sr_pair_pack	pair-wise	pack-unpack	`MPI_Sendrecv`
12	WinfencePut_aao	all-to-all	derived types	`MPI_Put/MPI_Win_fence`
13	WinfenceGet_aao	all-to-all	derived types	`MPI_Get/MPI_Win_fence`
14	PostStartPut_aao	all-to-all	derived types	`MPI_Put/ MPI_Win_post/start`
15	PostStartGet_aao	all-to-all	derived types	`MPI_Get/ MPI_Win_post/start`
16	WinfencePut_pair	pair-wise	derived types	`MPI_Put/MPI_Win_fence`
17	WinfenceGet_pair	pair-wise	derived types	`MPI_Get/MPI_Win_fence`
18	PostStartPut_pair	pair-wise	derived types	`MPI_Put/ MPI_Win_post/start`
19	PostStartGet_pair	pair-wise	derived types	`MPI_Get/ MPI_Win_post/start`

Table 3.1: Currently available codelets in ADCL in the predefined set of codelets `ADCL_FNCTSET_NEIGHBORHOOD` for the regular n-dimensional neighborhood communication and their corresponding attributes.

by ADCL differ in the number of simultaneous communication partners for each process (pair-wise or all-to-all), the data transfer primitives used (synchronous, asynchronous, one-sided communication) and the handling of non-contiguous data (derived data types or pack/unpack). It has been shown that close-to-optimal performance is delivered for a large number of platforms and network interconnects [79, 83].

Within the context of this work, the functionality of ADCL has been extended to collective communication operations and different methods to collect and evaluate empirical data have been implemented and analyzed. The details are described in sec. 3.4.

3.3.1 Using ADCL

ADCL has its own API, it does not use a compiler-based automatic code substitution or the performance interface of MPI since its goal is to tune any code fragment and to offer high level interfaces of application level collective operations which do not directly correspond to certain MPI function calls.

The main objects within the ADCL API describe the data, codelets and the MPI topology and are used to steer execution and optimization.

`ADCL_Vector` describes the data to be used during the communication. The user can for example allocate or register a data structure such as a vector or a matrix with the ADCL library, detailing the dimensions and extents of the structure, its basic data type, the number of components per grid point, the number of layers of halo cells and a pointer to the data array.

`ADCL_Function` is a codelet that implements a certain numerical kernel or communication pattern. The user can register its own codelets in order to utilize the ADCL runtime selection logic.

`ADCL_Fnctset` is a collection of ADCL functions providing the same functionality. ADCL includes a pre-defined function set `ADCL_FNCTSET_NEIGHBORHOOD` for the *n*-dimensional Cartesian neighborhood communication.

`ADCL_Attribute` is an abstraction for a particular characteristic of a codelet. Each attribute is represented by the set of possible values for this characteristic.

`ADCL_Attrset` is a collection of ADCL attributes.

`ADCL_Topology` provides in the MPI case a description of the process topology and neighborhood relations within the application.

`ADCL_Request` combines a vector object, a function set and a topology object. In analogy to the MPI case where the `ADCL_Request` represents a persistent communication object similarly to its MPI counterpart for sequential persistent requests, the ADCL request can be 'started', in this case using `ADCL_Request_start`. Call-site dependent optimizations can be realized by creating multiple ADCL requests.

The following code sample gives a simple example for an ADCL code, using a 2D neighborhood communication on a 2D process topology. It can be used within an existing parallelized program or as a guideline for an initial MPI parallelization where the user only has to add routines for data distribution and collection.

```
#include "mpi.h"
#include "ADCL.h"
```

```
/* Dimensions of the data matrix per process */
#define DIM0 1000
#define DIM1 1000

/* Number of layers of halo-cells */
#define HWIDTH 1

int main ( int argc, char **argv ) {

  int ndims      = 2;   /* number of dimensions */
  int vec_dims[2]={DIM0+2*HWIDTH, DIM1+2*HWIDTH};
                        /* extents of the data array */
  double matrix[DIM0+2*HWIDTH][DIM1+2*HWIDTH];
                        /* data array with halos to be communicated */
  int nc         = 1;   /* entries per grid point */
  int hwidth     = 1;   /* number of layers of halo cells */
  MPI_Comm cart_comm;

  /* Variables for the process topology information */
  int rank;              /* rank of the calling process */
  int size;              /* number of processes */
  int cart_dims[]={0,0}; /* number of nodes in each dimension */
  int periods[]={0,0};   /* no periodic grid */
  MPI_Comm cart_comm;    /* Cartesian MPI communicator */

  /* ADCL variables
  ADCL_Vector   adcl_vec;
  ADCL_Topology adcl_topo;
  ADCL_Request  adcl_request;

  int i;

  /* Initiate the MPI environment */
  MPI_Init ( &argc, &argv );
  MPI_Comm_rank ( MPI_COMM_WORLD, &rank );
  MPI_Comm_size ( MPI_COMM_WORLD, &size );

  /* Initiate the ADCL library */
  ADCL_Init ();

  /* Allocate a 2D matrix with ADCL */
  ADCL_Vector_allocate (ndims, vec_dims, nc, ADCL_VECTOR_HALO, hwidth,
     MPI_DOUBLE, matrix, &adcl_vec);
```

```
  /* Generate a 2-D process topology */
  MPI_Dims_create ( size, ndims, cart_dims );
  MPI_Cart_create (MPI_COMM_WORLD, ndims, cart_dims, periods, 0,
     &cart_comm);
  ADCL_Topology_create (cart_comm, &adcl_topo );

  /* Combine description of data structure, predefined function set and
     process topology */
  ADCL_Request_create (adcl_vec, adcl_topo, ADCL_FNCTSET_NEIGHBORHOOD,
     &adcl_request );

  /* Initialize matrix to zero including halo-cells */
  matrix_init ( vec_dims, cart_dims, matrix, cart_comm );

  /* Main application loop */
  for (i=0; i<NIT; i++ ) {
     /* Initiate neighborhood communication */
     ADCL_Request_start (adcl_request );
  }

  /* Output the resulting matrix */
  matrix_write ( matrix, cart_comm );

  ADCL_Request_free  ( &adcl_request );
  ADCL_Topology_free ( &adcl_topo );
  ADCL_Vector_free   ( &adcl_vec );
  MPI_Comm_free      ( &cart_comm );

  ADCL_Finalize ();
  MPI_Finalize ();
}
```

During `ADCL_Init` and `ADCL_Finalize`, internal data structures are initialized resp. freed including the predefined set of codelets such as `ADCL_FNCTSET_-NEIGHBORHOOD` for the n-dimensional neighborhood communication. Then, a two-dimensional, double precision matrix of dimensions `ndims` and extent `vec_dims` with one layer of halo cells is allocated as an `ADCL_Vector`. The `ADCL_Topology` object is based on a previously defined Cartesian communication `cart_comm`. `ADCL_Vector`, `ADCL_Topology` and the predefined set of codelets `ADCL_FNCTSET_NEIGHBORHOOD` are combined to an `ADCL_Request`. All necessary ADCL objects are now initialized. `ADCL_Request_start` replaces the calls to MPI. It executes the communication, controls the empirical optimization, and selects transparently to the user the optimal codelet.

With ADCL, the user is freed from the burden of buffer management or the creation of derived data types. All data management is done inside ADCL. At present, also buffered next-neighbor communciations have been implemented, so ADCL can decide based on the collected data, whether or not copying data is better then using derived data types. No more sophisticated guesses from the user are necessary regarding this matter.

The enhancements of ADCL within this thesis to support collective MPI communications required small changes to the API which are described in chapter 4.

3.3.2 Mode of operation

ADCL passes through the three phases of automatic tuning during the regular execution of the application, i.e. whenever *ADCL_Request_start* is called: *search*, *decision* and *production*.

The *search* is carried out during the first executions of the communication pattern in the application which is tuned. The simplest approach, a brute force search strategy, evaluates all available n_c codelets n_m times (for statistical reasons) whereas a search with selection logic would only evaluate some codelets. In iteration 1 to n_m, ADCL calls—transparent for the user—the first codelet and stores the execution times locally on each process. Although the execution times on each process may be very different, providing the measurements of all processes to all other processes would result in an exchange of large data volumes and an substantial increase in memory consumption and is therefore not feasible. An optional *selection phase* decides on the second codelet which execution times are measured during iterations $n_m + 1$ to $2 \cdot n_m$. After all n_c codelets have been tested, each process disposes of an array of $n_c \cdot n_m$ execution times. A more sophisticated search algorithm based on attributes is explained in sec. 3.3.3.

Assuming that the runtime environment produces reproducible performance data over the lifetime of an application, the brute force search is guaranteed to find the best performing codelet on a given platform. The major drawback of this approach is the time it might take to determine the fastest codelet. According to observations on various platforms, the library requires between 10 and 50 measurements per codelet in order to have enough data points for a statistical analysis. Taking into account that the library might have to test up to twenty different codelets, up to 1000 instances of the communication pattern might be required before the runtime selection logic comes up with a final decision. Although this does not necessarily translate into 1000 iterations in the application itself, since e.g. an iterative solver very often has multiple instances of the neighborhood communication within each iteration, adaptive applications

with varying problem sizes would require a significantly faster procedure in order for ADCL to become useful for this class of applications. So more efficient search strategies have to be applied which are described in sec. 3.3.3.

During the following *decision* phase, the measurements are analyzed mainly locally with a statistical method capable of handling outliers, i.e. "wrong" measurements. Only one collective communication is necessary to compute the global minimum. The statistical method decides on the codelet it considers to be the fastest, the so-called *winner codelet*. For a next-neighbor communication, for example, the statistical method would analyze the performance data for each of the 20 different codelets given in tbl. 3.1 and decide on the codelet it judges best-performing. In chapter 5, the current approach is compared to three other widespread statistical approaches, namely a standard interquartile range method, cluster analysis and robust statistics. The quality of the decision also depends on how accurately the collected data reflects the behavior of the application. That this is not automatically the case is shown in chapter 6 where alternative data collection methods have been implemented and analyzed.

Starting from iteration $n_c \cdot (n_m + 1)$, the winner codelet is used in *production* for all subsequent iterations.

An additional mode of operation called *preselection*, is less relevant in practice, but highly convenient for our research. The user specifies a certain codelet forcing ADCL to skip the search and decision phases and enter directly the production phase, so that this codelet is used during the whole runtime of the application. Using preselection, one can assess the performance of each codelet in long-term runs to determine which one is the best-performing codelet in the current setting. This allows e.g. to evaluate if the decision logic of ADCL made a correct choice or if it selected a codelet which does not deliver optimal performance. This is why we denote such a set of measurements where each codelet in a set of codelets is preselected and executed once in a long-term run as *verification run* throughout the rest of this work.

3.3.3 Overheads and Countermeasures: A Sophisticated Selection Logic and Historic Learning

Comparable to any runtime optimization library, ADCL introduces different types of overheads. As the search phase is integrated into the execution, non-optimal codelets are executed which causes the major part of the overhead. During production, the comparatively small calling overhead is present. Also, additional data structures have to be stored in main memory. ADCL does not write any data to hard drive, except if the user specifies to store the gathered performance data on disk.

The initial search overhead can be reduced by a more sophisticated algorithm or by historic learning, i.e. the exploitation of knowledge gathered in previous runs to avoid the search phase altogether. The user can also decide to use preselection after executing some tuning runs at the cost that changing parameters such as different nodes allocated by the batch scheduler are no longer taken into account.

Apart from the brute force search which evaluates all available codelets of a given communication pattern, ADCL can alternatively apply a heuristic search method based on attributes characterizing a codelet in order to speed up the runtime search procedure [79]. The heuristic is based on the assumptions that the fastest codelet for a given problem size on a given execution environment is also the codelet having 'optimal' values of the attributes and that the attributes are uncorrelated and can thus be tuned seperately one after the other. Therefore, the algorithm tries to determine the optimal value for each attribute used to characterize an codelet. After the hypothesis that a certain attribute value leads to better performance is confirmed by multiple codelets, the optimal value for this attribute is assumed to be found and the library removes all codelets not having the required value for the corresponding attribute from the function set and thus shrinks the list of available codelets. This can speed up the search tremendously as shown in [79].

The goal of historic learning is to exploit knowledge gained from previous executions to speed up the search process of ADCL without reducing the quality of the selection. The main reason is that for some clusters, the execution of some codelets during the search phase was so time-consuming that at the end of the run, the search phase still dominated the overall execution time. Moreover, applications with frequently and dynamically varying problem sizes, e.g. due to load-balancing or adaptive mesh refinement, can not spent a lot of time in the tuning phase. Currently, ADCL would probably often not finish tuning before the communication volumes change. As necessary prerequisites, Feki et al. [84] introduced a notion of similarity and a certain distance measure as well as a verifying procedure to eventually discard former performance data which is no longer valid in the current execution context.

Apart from these overheads, the performance gain of ADCL depends on firstly, if applicable, how good the winner codelet determined by ADCL is compared to the original MPI implementation and secondly, how long the application is running. The longer the production phase takes, the relatively smaller the initial overhead of the search phase gets.

3.4 Previously Missing Features and Contributions of This Work

The three research topics discussed in this work follow from the prior state of development of ADCL. The first one concerns ADCL's missing applicability to collective MPI communications and the contributions of this work to solve this problem are discussed in sec. 3.4.1. The other two are associated with the collection and evaluation of empirical data which the optimization process relies on: firstly, the problem of *outliers*, i.e. data points which deviate considerably from the others. They need to be checked for relevance, i.e. if an error of measurement occurred or if those "outliers" reflect a typical application behavior as explained in 3.4.2. This work analyzes multiple statistical methods to handle this problem. Secondly, the necessity of reliable empirical data that accurately predicts the performance of a codelet poses challenges discussed in sec. 3.4.3. Several new methods of data collection have been developed, implemented and analyzed for this work.

3.4.1 Area of Application

A shortcoming of ADCL in the past was its restriction to next-neighbor communications on Cartesian grids whereas a widely used class of MPI communication operations, namely collective operations, were not supported. Rabenseifner showed in a long-term study [24] that apart from point-to-point communications a significant amount of communication time is spent in the collectives `MPI_Alltoall` and `MPI_Allreduce`. Terry Jones concluded in a more recent study [85] that the time spent in collectives is predominantly divided among `barrier`, `allreduce`, `broadcast`, `gather` and `alltoall` and that the `alltoall` performance is vital to some codes. This set of operations is also of great value to the CFD community as e.g. the `alltoall` operation is the central part of FFTs needed to study transitional phenomena.

Since the performance of collective operations has a considerable influence on the scalability, many researchers tried to improve the efficiency of collectives [86, 76, 72, 87, 88] based on certain assumptions concerning the type of network and the message length. However, no single algorithm can lead to optimal performance in all possible scenarios. Even if the MPI implementation combines several algorithms using a heuristic switching technique depending on the message length, the best performance is not guaranteed. The MPI implementation itself will also use its own algorithm. Thus the user faces a large choice of possible alternative implementation without knowledge and/or resources to find the optimal one for his application. He/She has to decide

whether to use the internal MPI implementation of unknown quality or to efficiently implement one or more algorithms him-/herself suited for his/her application.

To support the user in his efforts, it is therefore of utmost importance to extend ADCL to support collective operations. The contributions described in chapter 4 take away complexity and work from the user and supply him/her with a valid tool which provides optimal communication performance for his/her application. It is shown that the native MPI implementation is not always the best choice. The extension required the introduction of a new ADCL object, the vector map object.

3.4.2 Outlier Handling

The decision logic is expected to come to a decision which codelet is the best-performing one based on the collected empirical data. This implies that for each codelet an estimated mean execution time is calculated. The codelet with the smallest estimated mean execution time is selected.

Measured communication times commonly differ from each other. If an unpredictable event, such as large network traffic from another application, occurs at the same time as the communication operation, the execution time will be a lot higher than expected. If one simply averages over all measurements for one codelet, the resulting mean is not characteristic for the actual performance of the codelet. Thus, it is mandatory to exclude or down-weigh the untypical data point before computing the estimated mean.

However, not all increased executions times are necessarily outliers. When codelets send large amounts of data at once, package drops at the network switch occur. Packages have to be retransmitted and cause increased execution times on a regular basis. If these data points are treated as outliers and are excluded or down-weighed, one ignores a fundamental property of the codelet and inevitably obtains an underestimated mean.

In short, an adequate outlier handling is crucial, otherwise the estimated means are incorrect. The decision logic then would arrive at a wrong conclusion leading to a performance penalty. It therefore has to be clarified, if a data point is truly an outlier or if it reflects a typical application behavior, and subsequently, how this data point should be handled when computing an estimated mean. Chapter 5 will present different statistical methods to tackle the outlier problem and contribute a thorough analysis of their quality in a variety of settings.

3.4.3 Reliable Performance Measurements

The second problem concerns the collection of empirical data. As a general rule, more data lead to a better statistic, less data create less overhead. However, if the measurements itself are of low quality and inhibit some (systematical) errors, more data does not improve the statistical result. Different influences lower the datas' reliability, especially when tuning parallel communications: on the one hand, subsequent measurements might influence each other, in particular when switching from one codelet to the next. On the other hand, adding synchronization disrupts the parallel execution. Additionally, the location of the calls to the timing routines in the source code can effect the quality of the data. System timers sometimes lack precision when execution times are very small. To sum up, a runtime optimization library needs to be able to predict the performance of a codelet based on only a few, but reliable data points to keep the search overhead low and select the best-performing codelet.

The problem of unreliable empirical data is illustrated in fig. 3.1. It shows the empirical data obtained from the collective MPI communication of a NAS FFT benchmark of class B (cf. 2.4.2) running on 32 processes on an SGI Altix with Intel MPI. Twenty measurements are performed for each of the eight different collective communication codelets c_0 to c_7. For a description of the codelets, the reader is refered to sec. 4.1.3. Each data point represents the averaged execution time of the communication operation over all processes. This data ($\Diamond$) and its estimated mean (red) are compared to the execution times of long-term simulations for each of the codelets (olive). For codelets c_1 (data points 21–40) and c_4 (data points 81–100) as well as to a lesser extent for c_3 (data points 41–60) and c_7 (data points 141–160), the empirical data is not representative of the overall performance. The data for the codelet c_0 contains to many outliers, so they are regarded as valid information and included in the calculation of the estimated mean.

Thus, apart from a inadequate outlier treatment, also a wrong assessment of the empirical data results in errors of the decision logic and subsequently performance losses.

3.4.4 Conclusions

MPI is sometimes playfully called the "assembler of parallel programming". To use MPI with all its possibilities is too complex and too rich in details for the average user. An example are the 20 possibilities how to realize a neighborhood communication pattern shown in tbl. 3.1. However, if one does not leverage MPI to its full extent, performance losses are inevitable. Since the amount of

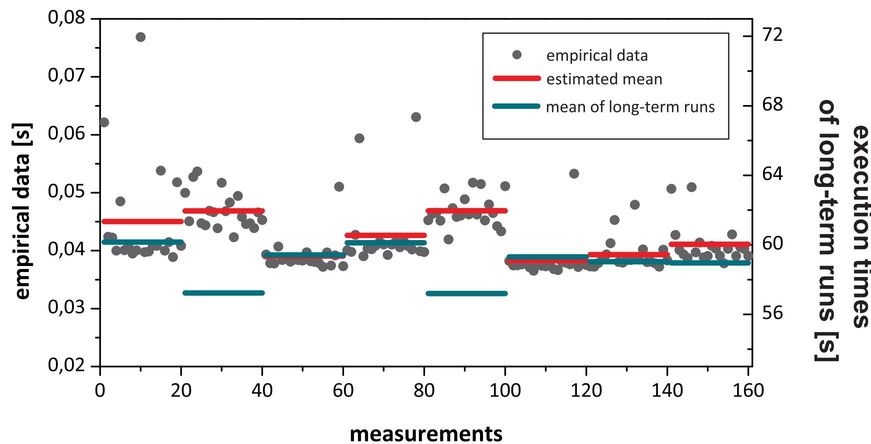


Figure 3.1: Wrong estimation of the mean due to non-representative empirical data.

computing time is often fixed, this results in simulations of lower resolution to make up for the slower runtime. But even with a detailed knowledge of MPI, it is not possible to write a single piece of code which will perform optimally across HPC systems and for different test cases and MPI implementations.

Thus, complexity needs to be reduced for the user. An abstract interface should allow him to simply describe his/her data and to which process it should be communicated. In addition, portable performance is required. Good performance for different settings (HPC system, test case, software, . . .) should be available without having to tune the code anew. This is what ADCL provides.

The state of development of ADCL was far from complete. One of the most important classes of MPI communications, namely collectives, were missing. This class has great impact on the scalability of large-scale simulations. It appears frequently in CFD applications, e.g. to calculate time steps, residuals or to distribute and collect data. Some applications even rely heavily on them, such as spectral methods. In the context of this work, the extension of ADCL to support this significant class of collectives is presented in chapter 4.

It also became clear from this section, that without proper data and the right statistical method, ADCL can not decide correctly on the best-performing codelet. This work creates a sound basis for the empirical data needed by the decision logic. Chapter 5 compares different approaches to handle the outlier problem and chapter 6 discusses how to obtain reliable empirical data.

Each HPC system is unique and the knowledge how to carry out optimizations is not readily available. Also it requires more knowledge than the average user

has. This makes it complicated for the user to apply optimizations to their own codes. ADCL helps to reduce the complexity while assuring an adequate performance. Details are hidden, but taken into account. Users who run their application on different clusters, HPC systems and at sites of collaboration partners profit from the performance portability.

4 Extending the Area of Application of ADCL

In this chapter, enhancements to the ADCL API are described in order to extend ADCL's functionality beyond Cartesian neighborhood communication, most notably to support MPI collective operations. The main challenge lied in the fact that up to now data description and information about the communication operation were intertwined. A new set of interfaces is developed and detailed which supports more generic communication operations, among them most MPI collective communication operations. They thus enable the optimization of one of the most widely used features of the MPI specification. Semantic as well as implementation aspects are discussed. An ADCL version of the NAS FT benchmark is implemented using the new interfaces for the all-to-all communication pattern and results for various problem sizes, number of processors and MPI libraries on several HPC systems are presented.

4.1 Semantics of new ADCL interfaces

The ADCL vector object as presented in sec. 3.3 serves two purposes: first, it allows to identify the buffer associated with a communication operation; second, it allows to perform an automatic data mapping of which portion of the data array is supposed to be transfered to which process. As an example for the latter the interface to allocate an ADCL vector,

```
int ADCL_Vector_allocate ( int ndims, int *dims, int nc, int comtype,
    int hwidth, MPI_Datatype dat, void *data, ADCL_Vector *vec )
```

contains the parameter `hwidth` which specifies the number of layers of halo cells that have to be transfered to the neighboring processes. Combined with the topology object based on a Cartesian communicator, the library automatically determines which elements have to be transfered to which process.

4.1.1 The ADCL vector-map object

Although this functionality is highly convenient, the corresponding API was limiting ADCL to Cartesian neighborhood communications, the original driving force of the library. To support further communication patterns, a set of new API interfaces is developed in this work, which allow to separate the description of the communication buffer and the mapping of which elements of the buffer have to be transfered to which process.

The new set of interfaces developed within this project tries to accommodate multiple goals:

1. allow for the definition of user defined functions as well as predefined set of codelets for common MPI communication operations, e.g. neighborhood communication, all-to-all, etc.
2. separate data management from the actual communication operations
3. allow for a light-weight description of data mappings and the automatic association with remote processes.

The new interface to create an `ADCL_Request` therefore distinguishes between five different objects: the vector object, the vector-map object, the topology object, the set of codelets and the request. The objects for attributes, attribute sets and codelets remain untouched. In the following, attention is devoted to the vector, vector-map and the request object.

The main purpose of the 'new' vector object is to define a data array that will be used later for communication. The interface allows to *register* an already existing or *allocate* a new multi-dimensional memory region, using the number of dimensions of the data array, extent of each array, number of elements of each data point in the array, and the basic MPI datatype. The vector does **not** specify which elements of the data array will be used in communication operations.

The vector-map object (or short 'vmap') allows to define which elements of the vector object have to be transfered to which process. This functionality does not have a counterpart in MPI, since it combines functionality often provided by the vector versions of the MPI collective operations such as `MPI_Gatherv`, `MPI_Scatterv` and derived MPI data types. Although the vmap object does not have the flexibility of the most generic MPI derived data type constructors such as `MPI_Type_create_struct`, it provides a much simpler and more user-friendly interface compared to the latter one, and covers the most common situations. Specifically, based on the vector and the vmap object, the ADCL library is able to construct the required derived MPI data types automatically for the end-user.

4.1.2 Extension of the ADCL Interfaces

In order to support a large variety of communication patterns, the information required for the MPI collective operations, the Cartesian neighborhood communication and user defined function sets is analyzed in the following.

For the collectives defined in the MPI standard and shown in tbl. 4.1 it is noticed that the parameters can be separated into three groups: information concerning the data (buffer, data type), concerning the process topology (communicator, root) and related to the communication pattern (element counts, reduction operation, array of element counts or displacements). As an example, for the `MPI_Bcast` interface, the information about the data consists of the `buffer` and the data `type`, `root` and the MPI communicator `comm` give information about the process topology and `count` is related to the communication pattern. As each of the parameters of the MPI collectives falls into one of these three groups, they form the basis of the new ADCL objects: information concerning the data is stored in the ADCL vector object, information concerning the process topology in the ADCL topology object and information related to the communication pattern becomes part of the new vmap object.

If one treats send and receive information separately, one obtains four different types of vmaps for the collectives. The interface of `MPI_Gather`, for example, specifies the same information—the amount of data to send/receive—once for the sender and once for the receiver. This combination is regarded as one vmap type called `all`. Together with an `inplace` type and the `halo` type for Cartesian neighborhood communication, this results in six different types of vmap objects. The constant `ADCL_VMAP_NULL` can be used for user-defined function sets that do not necessarily need a vmap object. Table 4.2 summarizes the different vector-map object types and the required information for each of them.

The types of vmap objects needed for each MPI collective are shown in tbl. 4.3. The interface for `MPI_Gatherv`, for example, contains as communication information the parameters `scount` for the sender as well as `rcounts` and `displs` for the receiver. This translates into the ADCL vmap types `all` and `list`. MPI allows the "in place" option for intra-communicators by passing the value `MPI_IN_PLACE` to `sbuf` at all tasks. Since the `scount` argument is then ignored, the other possibility for ADCL vmap types is `inplace` combined with `list`.

Due to the broad range of parameters required for various operations, different interfaces for different operations have been defined. The parameter `comtype` was originally used in the interfaces to allocate or register a vector and describes the type of vmap. It now becomes part of the interface to allocate the vmap object as `vmap_`*comtype*`_allocate`. The new interfaces for vmap, vector, topology and request creation for the example code from sec. 3.3 are now

collective	data information	communication information	topology information
`MPI_Bcast`	buffer, type	count	root, comm
`MPI_Gather`	sbuf, stype, rbuf, rtype	scount, rcount	root, comm
`MPI_Gatherv`	sbuf, stype, rbuf, rtype	scount, rcounts, displs	root, comm
`MPI_Scatter`	sbuf, stype, rbuf, rtype	scount, rcount	root, comm
`MPI_Scatterv`	sbuf, stype, rbuf, rtype	scounts, displs, rcount	root, comm
`MPI_Allgather`	sbuf, stype, rbuf, rtype	scount, rcount,	comm
`MPI_Allgatherv`	sbuf, stype, rbuf, rtype	scount, rcounts, displs	comm
`MPI_Alltoall`	sbuf, stype, rbuf, rtype	scount, rcount	comm
`MPI_Alltoallv`	sbuf, stype, rbuf, rtype	scounts, sdispls, rcounts, rdispls	comm
`MPI_Alltoallw`	sbuf, stypes, rbuf, rtypes	scounts, sdispls, rcounts, rdispls	comm
`MPI_Reduce`	sbuf, rbuf, type	count, op	root, comm
`MPI_Allreduce`	sbuf, rbuf, type	count, op	comm
`MPI_Reduce_ Scatter_block`	sbuf, rbuf, type	rcount, op	comm
`MPI_Reduce_ scatter`	sbuf, rbuf, type	rcounts, op	comm
`MPI_Scan`	sbuf, rbuf, type	count, op	comm
`MPI_Exscan`	sbuf, rbuf, type	count, op	comm

Table 4.1: Overview of collectives. Interface parameters are divided into data, communication and topology-related information (s — send, r — recv).

```
ADCL_Vmap_halo_allocate ( int hwidth, ADCL_Vmap *vmap );
ADCL_Vector_allocate ( int ndims, int *dims, int nc, MPI_Datatype dat,
                       void *data, ADCL_Vector *vec );
ADCL_Topology_create ( MPI_Comm comm, int, root, ADCL_Topology *topo );
ADCL_Request_create ( ADCL_Vector vec, ADCL_Vmap vmap,
                      ADCL_Topology topo, ADCL_FNCTSET_NEIGHBORHOOD,
                      ADCL_Request *request );
```

Special attention has to be attributed to the data types which naturally belong to the vector object. However, also the request object has data types since in case of the neighborhood communication, the vector object contains the basic data type whereas for the request derived types are constructed which depend on the topology information (size), data information (dimensions and extent of the data array, *nc*) and communication information (hwidth). This means that for collectives the data types from the vector object have to be copied to the request object. New variables are introduced in the request object which specify the number of MPI data types to be sent or received. For the copy operations of the data types and the initialization of the new variables, the functions `ADCL_basic_init` and `ADCL_basic_free` are implemented.

4.1.3 The new function sets

For supporting MPI collective operations within ADCL, one ADCL codelet is a wrapper around the native MPI collective provided by the MPI library. This codelet is denoted as *encapsulated codelet*. Additionally, a variety of algorithms are implemented which perform the collective communication based on point-to-point communications. Inside the MPI library, the collective function is likewise performed by one algorithm as a sequence of point-to-point communications, but it is up to the MPI library which algorithm it uses and in case of vendor MPI libraries not known to the user.

Vmap comtype	Parameters
inplace	-
halo	hwidth
all	count
reduce	count, op
list	counts, displs
redscatter⋆	counts, op

Table 4.2: Types of ADCL vmap objects (⋆ - not implemented as only needed for `MPI_Reduce_scatter`).

collective	vmap type for send (& receive) vectors		
MPI_Bcast	all		
MPI_Gather	all & all	or	inplace & all
MPI_Gatherv	all & list	or	inplace & list
MPI_Scatter	all & all	or	all & inplace
MPI_Scatterv	list & all	or	list & inplace
MPI_Allgather	all & all	or	inplace & all
MPI_Allgatherv	all & list	or	inplace & list
MPI_Alltoall	all & all	or	inplace & all
MPI_Alltoallv	list & list	or	inplace & list
MPI_Alltoallw	list & list	or	inplace & list
MPI_Reduce	reduce (2x)	or	inplace & reduce
MPI_Allreduce	reduce (2x)	or	inplace & reduce
MPI_Reduce_ Scatter_block	reduce (2x)	or	inplace & reduce
MPI_Reduce_ scatter	redscatter (2x)	or	inplace & redscatter
MPI_Scan	reduce (2x)	or	inplace & reduce
MPI_Exscan	reduce (2x)	or	inplace & reduce

Table 4.3: Types of ADCL vmap objects for the MPI collectives.

New predefined function sets for allreduce, allgatherv and all-to-all have been added. ADCL provides five codelets for the allreduce operation. One is the encapsulated native MPI_Allreduce, the four others are based on send-receive operations: linear (reduce to root and broadcast with own implementations), nonoverlapping (reduce and broadcast with MPI implementation), recursive doubling algorithm as used in MPICH2 [89] for small and intermediate size messages and ring.

For allgatherv, there is the native MPI implementation, linear (gatherv to root and broadcast), recursive-doubling as used in MPICH2 [89], Bruck (a variation of the All-to-all algorithm described in [90]), neighbor exchange (adapted from allgather algorithm described by Chen et.al. in [91]) and ring.

The all-to-all function set consists of eight codelets, here numbered for reference in sec. 4.2: the native MPI_Alltoall (c0), linear_sync (c1), pairwise (c2), pairwise_excl (c3), linear (c4) and Bruck's Algorithm [90] with a minor modification as used in MPICH2 [89], which restricts the number of messages. In our case block sizes of 2, 4 and 8 (c4–c7) are used.

No special effort has been invested to tune the different codelets as of today. In a long-term, it is planned to add the flexibility for supporting various data transfer

primitives (blocking, non-blocking, one-sided) and various methods to handle non-contiguous data, similarly to the Cartesian neighborhood communication.

4.2 Performance Evaluation

To compare the codelets of ADCL with the native `MPI_Alltoall`, the MPI FFT Benchmark of the NAS Parallel Benchmarks 3.0 presented in sec. 2.4 is used. However, there is no intent to compete with FFTW or similar software packages. The purpose is not to gain maximum performance for this special application, but to obtain a valid tool for the optimization of all collectives.

4.2.1 Integration of ADCL

The modified code of the FT benchmark is depicted in fig. 4.1 and 4.2. In the main program, after the call to the benchmark's `setup()` routine, `ADCL_Init` is called and the ADCL data structures are build: a vmap object for the all-to-all communication, vector objects for u_1 and u_2 are registered and a request object is allocated. A switch is set to false to avoid counting the first call to `fft()`. It is set to true right before the main loop. During the computation of the FFT in the subroutine `transpose2_global` the call to `MPI_Alltoall` is replaced by an `ADCL_Request_start`. After the call to `print_timers()` the ADCL objects are deregistered or deallocated and `ADCL_Finalize` is called. The header file `ADCL.inc` is included in the main program as well as in the subroutine.

The following variable declarations have been added in `global.h`:

```
      integer adcl_topo, adcl_vmap, adcl_svec, adcl_rvec, adcl_request
      logical use_adcl

      common /adcl/ adcl_topo, adcl_vmap, adcl_svec, adcl_rvec,
     >    adcl_request, use_adcl
```

4.2.2 Setup

The test systems used are the *Laki* cluster with InfiniBand interconnect, a Cray XT5m and a NEC-SX8 installation at HLRS, the SGI Altix at LRZ Munich and the Blue Gene/P system at the Supercomputing System Jülich. Runs were executed in the virtual node mode, i.e. every core ran an MPI process. Within a single batch job, three set of runs are executed. Each set consists of 9 runs, one for each of the 8 codelets presented in sec. 4.1.3 for the all-to-all operation and one without ADCL, with 200 FFT iterations.

```
 program ft
 include 'ADCL.inc'
c  FT: further includes and declarations
  call MPI_Init(ierr)
  call ADCL_Init(ierr)
c FT: timing and setup

c set up ADCL data structures
  call adcl_topology_create ( MPI_COMM_WORLD, 0, adcl_topo, ierr )
  call adcl_vmap_alltoall_allocate( ntdivnp/np, ntdivnp/np,
 >     adcl_vmap, ierr )
  call adcl_vector_register ( 1, ntdivnp, 0, dc_type, u2,
 >     adcl_svec, ierr )  ! send vector
  call adcl_vector_register ( 1, ntdivnp, 0, dc_type, u1,
 >     adcl_rvec, ierr )  ! receive vector
  call adcl_request_create_generic ( adcl_svec, adcl_vmap,
 >     adcl_rvec, adcl_vmap, adcl_topo, ADCL_FNCTSET_ALLTOALL,
 >     adcl_request, ierr )

c disable adcl while problem is ran once for benchmarking reasons
  use_adcl = .false.
c run problem
  use_adcl = .true.

c main loop
  do iter = 1, niter
     call evolve(...)
     call fft(...) ! calls transpose_xy_z which calls transpose2_global
  end do

c FT: verification and output

c free ADCL objects
  call adcl_request_free ( adcl_request, ierr )
  call adcl_topology_free ( adcl_topo, ierr )
  call adcl_vector_deregister( adcl_svec, ierr )
  call adcl_vector_deregister( adcl_rvec, ierr )
  call adcl_vmap_free (adcl_vmap, ierr )

  call ADCL_Finalize(ierr)
  call MPI_Finalize(ierr)
  end program FT
```

Figure 4.1: ADCL implementation of the main program of the FFT NAS Parallel Benchmark.

```
  subroutine transpose2_global(xin, xout)
  include 'ADCL.inc'
c FT: further includes, declarations and timing
  call adcl_request_start ( adcl_request, ierr ) ! replaces
c FT:  call mpi_alltoall(xin, ntdivnp/np, dc_type,
c FT: >                  xout, ntdivnp/np, dc_type,
c FT: >                  commslice1, ierr)
c FT: timing
  end
```

Figure 4.2: ADCL implementation of the subroutine which communicates the data of the FFT NAS Parallel Benchmark.

4.2.3 Results

This work follows a two-step procedure to analyze the results. At first, the execution time of the winner codelet is compared to that of the encapsulated native MPI_Alltoall. Secondly, the overhead caused by ADCL is taken into account and the execution time of the ADCL winner codelet is compared to the benchmark results without ADCL.

In the following, the subscript w refers to the winner codelet, n to the encapsulated native MPI_Alltoall and o to the original implementation without ADCL. The mean $\bar{t}_k = \frac{1}{n_r}\sum_{i=1}^{n_r=3} t_{k,i},\ k \in \{w, n, o\}$ is executed from the execution times $t_{k,i}$ of the $n_r = 3$ verification runs. The degree of dispersion about the mean is expressed by the uncertainty $u_k = \frac{s_k}{\sqrt{n_r}}$ where

$$s_k = \sqrt{\frac{1}{n_r - 1}\sum_{i=1}^{n_r}(t_{k,i} - \bar{t}_k)^2}$$

is the sample standard deviation.

For $k \in \{n, o\}$, the possible gains in percent are given by

$$g_k(\bar{t}_k, \bar{t}_w) = \frac{\bar{t}_k - \bar{t}_w}{\bar{t}_k} \cdot 100$$

with uncertainty

$$\begin{aligned} utot_k &= \sqrt{\left(\frac{\partial g_k}{\partial t_k} \cdot u_k\right)^2 + \left(\frac{\partial g_k}{\partial t_w} \cdot u_w\right)^2} \\ &= \sqrt{\left(\frac{t_w}{t_k^2} \cdot u_k\right)^2 + \left(\frac{u_w}{t_x}\right)^2} \cdot 100\,. \end{aligned}$$

When comparing the winner codelet to the encapsulated native `MPI_Alltoall`, in 22 out of 42 test cases, i.e. more than 50%, the encapsulated all-to-all was not the best implementation as shown in fig. 4.3. In other words, there was another ADCL codelet, i.e. one algorithm based on point-to-point communications, that outperformed without any tuning the native MPI collective! Among them were 6 cases in which ADCL performed better (more than 10%). It is important to note that 7 out of 8 codelets provided by ADCL performed best in at least one test case. This shows the value of ADCL: firstly, one can not trust in the performance of the MPI implementation. Secondly, it clearly demonstrates the need for automatic tuning since no single codelet performs best in all cases.

Secondly, the overhead caused by ADCL is taken into account and the execution time of the ADCL winner codelet is compared to the results of the original benchmark without ADCL. The possible gains and losses using ADCL are depicted in fig. 4.4. Except for the test cases sgi_altixmpi_8B ($g_o = -1.32\%$, $utot_o = 1.08\%$), sgi_altixmpi_32B ($g_o = -1.56\%$, $utot_o = 1.31\%$) and jugene_vn_512C ($g_o = -0.05\%$, $utot_o = 0.01\%$), ADCL performs as good or better than the original version without ADCL. There are no rules identifiable in which test case the use of ADCL is strongly advisable. This once more strengthens the demand for an automatic tuning system to be able to switch to a different codelet.

4.3 Conclusion

This chapter of the thesis analyzes how to extend the auto-tuning library ADCL. It describes the goals, semantics and realization of a set of new interfaces. With the vector-map object developed in the context of this work, data management and communication information are separated to allow for new communication patterns or user-defined functions. Predefined function sets for allreduce, allgatherv and alltoall have been implemented. The approach has been validated by analyzing the requirements of collective operations and user-defined functions.

Although the initial implementation for the predefined function sets does not support various data transfer primitives or methods to handle non-contiguous data, the results obtained for the NAS FT benchmark with its all-to-all communication pattern showed that the native MPI implementation has performance deficits and in all but three cases an equal or superior performance could be achieved when using ADCL. This makes ADCL an useful tool for vendors and administrators. The former can use ADCL to verify the performance of their native MPI collective implementation. The latter get assistance when they have to decide which MPI to recommend to users.

test case	winner codelet	gain g_n [%]	$utot_n$
laki_impi_8A	C7	0.67	0.19
laki_impi_8B	C7	2.03	0.51
laki_impi_128C	C2	0.05	1.07
laki_impi_512C	C3	**35.85**	0.87
laki_ompi_8A	C4	1.83	1.23
laki_ompi_8B	C1	0.34	0.50
laki_ompi_32B	C3	1.74	0.96
laki_ompi_32C	C3	0.46	1.88
laki_ompi_128C	C2	1.52	0.89
laki_ompi_512C	C3	**34.31**	2.21
sgi_altixmpi_256C	C2	**20.27**	2.76
sgi_impi_8A	C7	3.56	0.25
sgi_impi_8B	C4	**22.44**	8.54
sgi_impi_32B	C4	4.92	0.29
sgi_impi_128C	C4	**11.32**	1.47
sgi_impi_256C	C1	4.16	5.13
sgi_ompi_8B	C4	2.16	0.86
sgi_ompi_32B	C3	**56.23**	23.91
cray_8A	C7	0.70	0.15
cray_8B	C7	1.14	0.34
cray_128C	C3	0.38	0.71
cray_256C	C3	6.82	6.62
sx8_8A	C7	0.74	0.07
sx8_8B	C4	0.32	0.07
sx8_32B	C4	1.35	0.50
jugene_vn_8A	C5	1.84	0.01
jugene_vn_32B	C1	2.14	0.00

Figure 4.3: Overview of test cases with other winning codelets than the encapsulated native MPI implementation C0.

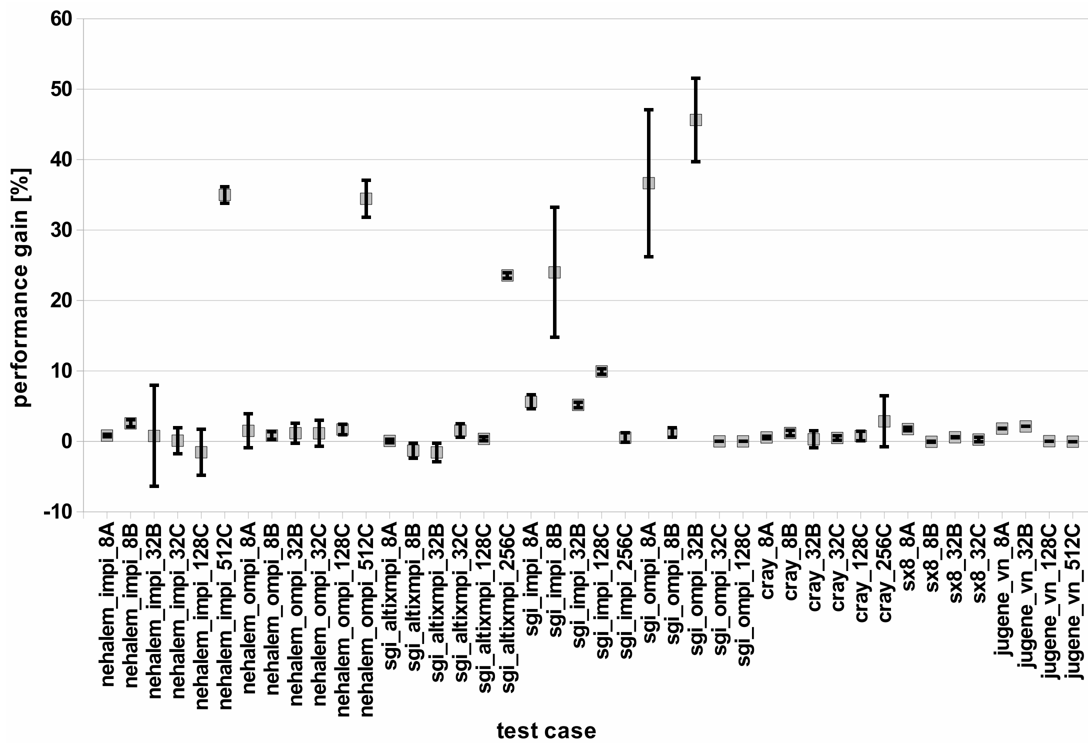


Figure 4.4: Performance gains and losses g_o in percent when comparing the total runtime with ADCL's winner codelet for the all-to-all communication to the one of the original NAS FFT benchmark version without ADCL. The error bars show the uncertainty $utot_o$.

5 Outlier Analysis and Evaluation of Different Decision Algorithms

This chapter evaluates the possible overheads of ADCL as an empirical optimization framework and gives recommendations how to minimize them. The two main sources for overheads are the search phase as non-optimal codelets are tested and eventually the production phase if the decision algorithm selects a non-optimal codelet[1]. A rule of thumb is established on how many data points are needed for each codelet and four different decision algorithms are compared.

5.1 Outliers in Performance Data

When an adaptive software component is employed to select the best-performing codelet at runtime, an estimated mean is calculated for each codelet from the empirical data. This estimated mean constitutes the basis for the decision to be taken and strongly depends on a reasonable handling of outliers. An outlier has mostly been defined in a strict statistical context as a data point "... *that deviates so much from other observations as to arouse suspicion that it was generated by a different mechanism*" [92]. In general, outliers may occur as a result of wrong models under study, systematic errors or faulty data, i.e. errors in measurement. The sample mean $\frac{1}{n_m}\sum_i y_i$ of n_m data points, for example, is not an an adequate estimator of the mean when outliers are present. Just one value $y_i \to \infty$ makes it arbitrarily large and thus completely meaningless.

When auto-tuning communication operations, the process of detecting and eventually removing outliers is greatly complicated by the fact that the types and quantities of outliers depend on the network interconnect and the nodes assigned to the job by the batch scheduler. In this work, four different statistical methods used for handling outliers are evaluated:

[1]The third source of overhead, the overhead of ADCL itself, is not always possible to state. There is not necessarily an equivalent implementation of the communication pattern of the application as codelet in ADCL. For example for the application code Euler3D used in this chapter, the data exchange in the application is based on temporary buffers, while ADCL uses derived data types.

- a **heuristic** derived from the trimmed mean value, where data with a large value are considered outliers and, if not numerous, removed,
- a **standard interquartile range method**, which constitutes a simple, common statistical approach,
- **cluster analysis**, where the data is partitioned into subsets based on similarity and small, distant subsets are excluded and
- **robust statistics**, a statistically justified procurement in which suspicious data is down-weighted rather than removed.

The correctness of the decisions with each of these statistical approaches are verified using performance data in a variety of scenarios over two fundamentally different network interconnects: a highly reliable InfiniBand network and a Gigabit Ethernet network having a larger variance in the performance.

Various types of network interconnects have dissimilar characteristics and produce therefore different types and quantities of outliers. This renders the distinction between systematic errors and faulty data particularly difficult.

High performance network interconnects such as InfiniBand or Myrinet configured in a fully non-blocking mode will not get congested for the vast majority of applications, i.e. the links are unlikely to carry so much data that a deterioration in the quality of service occurs. Such a network will produce highly reliable and repetitive performance data. One might still observe outliers in performance data over such high performance networks. They will however not be due to networking issues but external sources such operating system jitter [67].

A Gigabit Ethernet switch providing the full bisection bandwidth required for the given number of cluster nodes offers similar overall characteristics as the networks described above. However, depending on the quality of the switch, the performance data might show higher variations compared to an InfiniBand or Myrinet network. These variations make the identification of outliers more complex.

Another category of cluster networks are hierarchical networks, where (relatively) small switches are used to connect the nodes, and an additional layer of switches is used to connect these smaller, lower level switches. Due to technical or financial reasons, switches are often constructed using a fat tree topology with reduced uplink bandwidths. Examples are InfiniBand switches configured in a 2:1 blocking mode as for instance in the Thunderbird cluster at Sandia National Laboratories [93], or hierarchies of Gigabit Ethernet switches configured in many off-the-shelf clusters in industry and institutes or as the secondary network.

Communication patterns on such hierarchical networks can produce data points which could be considered by a trivial outlier detection algorithm as an invalid

data point. Figure 5.1 shows the execution time per instance of two different codelets for a neighborhood communication run across a hierarchical Gigabit Ethernet network. Both codelets produce a significant number of data points at the 200,000 μs range, the total number however varies greatly. They are due to congestion and packet drops at the higher level Gigabit Ethernet switch. Nevertheless, these data points in both cases contain valuable information regarding the performance of the codelet due to the systematic nature of their existence and can not be considered as outliers. Excluding them would lead to a wrong decision of the runtime logic, but, as stated earlier, ignoring the fact that performance data can contain outliers would lead to a wrong decision for the first two classes of networks. True outliers are located at 300,000 μs and above.

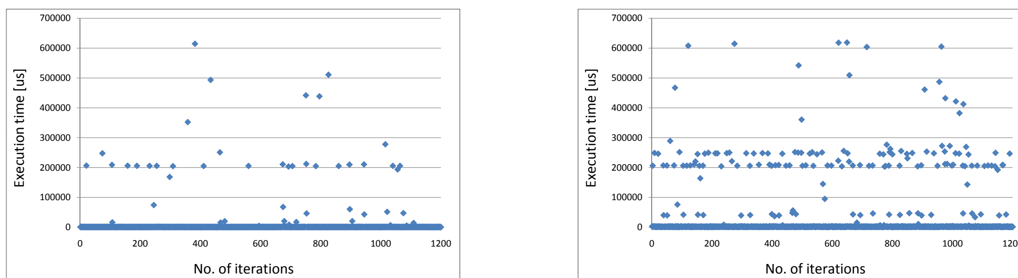


Figure 5.1: Performance data of two different codelets collected on a hierarchical network on process 0. Congestion and packet drops at the higher level Gigabit Ethernet switch cause relevant data points at the 200,000 μs range which has to be taken into account whereas data points above 300,000 μs can be regarded as outliers.

5.2 Techniques for Performance Data Evaluation

In this work, four different statistical methods used for handling outliers are considered, namely a heuristic derived from the trimmed mean value, a standard interquartile range method, cluster analysis and a method using robust statistics. Based on per-process performance data of each codelet, each technique has to be able to exclude incidental outliers and generate a consistent result across all processes while being at the same time computation and communication wise inexpensive in order to minimize the disruption of the application.

For each codelet c on each parallel process p, n_m measurements are performed and denote the execution time of the m-th measurement by $t(c,p,m)$.

For the first three methods presented, outliers, i.e. measurements not fulfilling a certain condition $\mathcal{C}$, are removed from the data set. This leads to a filtered subset S_f of measurements,

$$S_f(c,p) = \{t(c,p,m) \,|\, t(c,p,m) \text{ fulfills } \mathcal{C}\}\,, \tag{5.1}$$

with cardinality $n_f(c,p)$. Then, common to all methods, the performance data of each method is analyzed locally on each processor and characterized by an estimate of the mean value. This can be the local average execution time

$$\hat{\mu}(c,p) = \frac{1}{n_m}\sum_m t(c,p,m) \tag{5.2}$$

or its filtered counterpart

$$\hat{\mu}_f(c,p) = \frac{1}{n_f}\sum_{m\in S_f(c,p)} t(c,p,m). \tag{5.3}$$

After a global reduction, a maximum average execution time for each algorithm over all processes is determined. Its minimum is chosen as the best-performing algorithm.

5.2.1 Heuristic approach

This procedure based on a heuristic was first presented in [79] and has been applied in ADCL since. It is motivated by fig. 5.1 and relies on two parameters, namely a bound b, which defines a measurement to be an outlier and the maximum number of accepted outliers $n_o^{\max}$. This implies that the condition $\mathcal{C}$ which defines a measurement as outlier is defined as

$$\left\{t(c,p,m) \,|\, t(c,p,m) \leq b \cdot \min_m t(c,p,m)\right\}.$$

Outliers from below are practically non-existent. Measurements are regarded faulty and removed, if in general parlance there are up to $n_o^{\max}$ outliers, and an estimated mean is computed from the remaining ones. If there are more than $n_o^{\max}$ outliers, it is assumed that they contain valuable information and they are included in the computation of the estimated mean.

More precisely, the number of outliers $n_o(c,p) = n_m - n_f(c,p)$ is computed locally for each algorithm in addition to eq. (5.2) and (5.3).

After the global reduction for each codelet, the maximum average execution time

$$\hat{\mu}(c) = \max_p \hat{\mu}(c,p),$$

the maximum average execution time considering only filtered data

$$\hat{\mu}_f(c) = \max_p \hat{\mu}_f(c,p) \tag{5.4}$$

and the maximum number of outliers

$$n_o(c) = \max_p n_o(c,p)$$

for each codelet is determined. Finally, the maximum execution time including or excluding outliers is selected by

$$\hat{t}_h(c) = \begin{cases} \hat{\mu}_f(c) & \text{if } n_o(c) \leq n_o^{\max} \\ \hat{\mu}(c) & \text{otherwise} \end{cases}$$

depending on whether the maximum number of outliers is exceeded or not. The codelet c' fulfilling $\hat{t}_h(c') = \min_c \hat{t}_h(c)$ is chosen as the best one.

This procedure is related to the trimmed mean, which deletes a certain percentage of observations from each end of the data and then computes the mean in the usual way. Despite the fact that the trimmed mean is a simple robust estimator of location (see 5.2.4 for more details on robust methods), its application in this context is not beneficial since outliers with a very small execution time hardly appear and have much less impact on the mean value than outliers with large execution times. Nonetheless, without preceding tests on an architecture a non-optimal selection of the parameters b and $n_o^{\max}$ and, consequently, of the best-rated codelet, is possible. The method is thus only as good as the user's knowledge about the right choice of these twiddle factors.

5.2.2 Standard Interquartile Range Method

A common approach to remove outliers from a data set consists in using the interquartile range IQR which is defined as the difference between the third and first quartile [94]. A quartile, in turn, is any of the three values that divides a sorted data set into four parts of equal size. The first quartile Q1 marks the end of the first 25% of data, the third quartile Q3 the beginning of the last 25%. Every value being smaller than $Q1 - 1.5\,IQR$ or larger than $Q3 + 1.5\,IQR$ is regarded as outlier.

Outliers are discarded by setting $\mathcal{C}$ to

$$\mathrm{Q1} - 1.5\,\mathrm{IQR} \leq t(c,p,m) \leq \mathrm{Q3} + 1.5\,\mathrm{IQR}$$

in (5.1).

The filtered mean (5.3) is computed locally and after a global reduction, the maximum filtered average execution time for each codelet (5.4) is calculated. The best-performing algorithm c' is chosen as the one fulfilling

$$\hat{\mu}_f(c') = \min_c \hat{\mu}_f(c).$$

5.2.3 Cluster Analysis

Cluster analysis [95] is a common name for a variety of mathematical methods to partition a data set into subsets called clusters such that similar objects are grouped together. The similarity or dissimilarity is expressed by a distance measure, commonly the Euclidean distance. The average linking method employed here belongs to the group of hierarchical clustering methods which agglomeratively build up a hierarchy. The visual representation of the hierarchy in a treelike structure is denoted as dendrogram. Since outliers become small and isolated clusters, cluster analysis provides a natural mean to decide whether a point is an outlier or not. This eliminates the need to specify a bound b as in sec. 5.2.1, but a proper choice of the parameter $n_o^{\max}$ remains.

For the analysis of performance data, starting from the top of the dendrogram, those clusters are iteratively removed, which have little similarity to the others and which in total do not exceed the number of accepted outliers $n_o^{\max}$. This procedure ensures that the most isolated clusters are removed first. The current implementation utilizes the average linking method of the Open Source package Cluster 3.0 [96] with an Euclidean distance measure without prior standardization. After having removed the outliers, the local filtered means and global maximum average execution times are calculated as described sec. 5.2.2.

5.2.4 Robust statistics

Robust statistics [97, 98, 99] provide a statistically justified way of procurement. Its development was triggered due to problems where the clear decision whether to keep or reject a data sample can not be decided easily. Downweighting dubious observations is considered more appropriate than removing them completely from the data set, i.e. outliers are handled automatically and appropriately by the method. This distinguishes robust statics from all methods mentioned so far which require the removal of outliers.

Real data can deviate from the often assumed normal distribution in classical methods. This departure can manifest itself in the form of longer tailed or skewed distributions. Therefore, robust parametric statistics tend to rely on replacing the normal distribution with the longer-tailed t-distribution with low degrees of freedom or with a mixture of two or more distributions. Within the framework of this analysis, the first approach is applied.

The sample data recorded for the n units, i.e., the measurements of execution time $t(c', p', m)$ for one algorithm on one processor, are denoted for the sake of simplicity by $y_1, y_2, \ldots, y_n$. Let $\mathcal{N}(\mu, \sigma^2)$ denote the normal distribution with mean μ and variance σ^2. Then the normal model assumes

$$y \overset{\text{ind}}{\sim} \mathcal{N}\left\{\mu(\theta), \sigma^2(\phi)\right\}, \tag{5.5}$$

where μ is a function of known form indexed by an unknown parameter θ, and $\sigma^2(\phi)$ is the variance of known form indexed by a unknown parameters ϕ. The t model replaces the normal distribution assumption (5.5) by

$$y \overset{\text{ind}}{\sim} t\left\{\mu(\theta), \psi(\phi), \nu\right\},$$

where $t\{\mu, \psi, \nu\}$ denotes the t-distribution with location parameter μ, scale parameter ψ and ν degrees of freedom. The meaning of ν will be explained later.

A maximum likelihood estimation[2] applied to make inferences about parameters of the underlying t-distribution from the given data set analogous to those for the normal model (5.5).

Let $f(y_i \mid \vec{\xi})$ be the logarithm of the density function for $t\{\mu(\theta), \psi(\phi), \nu\}$. Maximizing the log-likelihood function

$$l(\theta, \phi, \nu) = \sum_{i=1}^{n} f(y_i \mid \vec{\xi}), \tag{5.6}$$

with respect to $\vec{\xi} = (\theta, \phi, \nu)$ yields maximum likelihood estimates $\hat{\vec{\xi}} = (\hat{\theta}, \hat{\phi}, \hat{\nu})$. Since the density function of the univariate t-distribution is

$$p(y \mid \vec{\xi}) = \frac{\Gamma\left(\frac{\nu+1}{2}\right)\psi^{-1/2}}{\sqrt{\pi\nu}\,\Gamma\left(\frac{\nu}{2}\right)} \cdot \left(1 + \frac{(y-\mu)^2}{\psi\nu}\right)^{-(v+1)/2},$$

[2]The maximum likelihood estimation is a statistical method widely used to identify estimates for the parameters of a statistical model by fitting it to the data.

where Γ is the Gamma function, the log-density function for each component y_i is

$$\begin{aligned} f(y_i|\vec{\xi}) &= -\frac{\nu+1}{2}\ln\left[1+\frac{(y_i-\mu(\theta))^2}{\nu\psi(\phi)}\right] \\ &\quad -\frac{\ln(\psi(\phi))}{2}+g(\nu)+\tilde{c}, \end{aligned}$$

where

$$g(\nu)=\ln\left[\Gamma\left(\frac{\nu+1}{2}\right)\right]-\frac{\ln(\nu)}{2}-\ln\left[\Gamma\left(\frac{\nu}{2}\right)\right], \tag{5.7}$$

and

$$\tilde{c}=-\frac{1}{2}\ln(\pi).$$

As explained in [100], for $\nu < \infty$, maximum likelihood estimation of θ and certain functions of ϕ are robust in the sense that outlying cases with large Mahalanobis distances $\delta_i^2 = (y_i-\mu)^2/\psi$ are down-weighted. The degree of down-weighting of outliers depends reciprocally on ν. If ν is fixed a priori at some reasonable value (degree of freedoms between 4 and 6 have often been useful in practice), it is a robustness tuning parameter. For larger data sets, ν can also be estimated from the data by maximum likelihood, yielding an adaptive robust procedure.

In this work the simplex algorithm of Nelder and Mead [101] was applied for the sake of simplicity to maximize (5.6). It is part of the GNU Scientific Library. For constant ν, $g(\nu)$ in (5.7) does not enter the maximization process and was therefore omitted. The median, the second quartile (see sec. 5.2.2), was chosen as an estimate of location and a multiple of the Median of Absolute Deviations (MAD)

$$\text{MAD}(y)=\text{median}_i|\,y_i-\text{median}_p(y_p)\,|$$

as a scale estimate. A factor of 1.483 is most commonly used.

After obtaining the estimate mean value μ, the same steps as in sec. 5.2.2 were applied to select the best codelet.

5.2.5 Complexity estimates

As mentioned earlier, the methods discussed within this section will be applied within a runtime library. This necessitates to minimize the computational costs of the algorithms handling outliers. Table 5.1 gives an overview about the complexity estimate for each of the four statistical approaches, with n_m being the number of measurements per codelet.

Statistical approach	Complexity
Interquartile range	$O(n_m \log n_m)$
Heuristic approach	$O(n_m)$
Cluster analysis	$O(n_m^3)$
Robust statistics	exponential in worst case, a lot faster in most real scenarios

Table 5.1: Complexity estimate for each of the four statistical approaches

5.3 Performance Evaluation

The behavior of each statistical method presented in the previous section is evaluated for different scenarios, i.e., whether the decision of the statistical method based on a small number of measurements is comparable to a ranking obtained by long-term runs. For this, performance data is generated using the structured Finite Volume solver Euler3D presented in sec. 2.4.1.

5.3.1 Integration of the ADCL

It took only half a day of work to add ADCL to Euler3D. The necessary code changes are described in the following.

The original MPI implementation exchanged the primitive variables and the left and right states for the Riemann Problem using the following scheme: the data from the halo cells is copied to buffers, the buffers are exchanged step by step in x-, y- and z-direction using `MPI_Isend/MPI_Recv/MPI_Waitall` and finally the data from buffers is copied back into halo cells. The data for the Riemann problem are stored in three arrays `xPrimLR`, `yPrimLR` and `zPrimLR` where the first dimension is used to distinguish the location of the halo cells (left/right, front/back or up/down). Thus, one iteration of the solver requires seven neighborhood communications to update the primitive variables as well as the states for the Riemann solver in the halo cells.

The first change was a change of data structure. At present, ADCL requires that communication arrays are stored contiguously in main memory, i.e. instead of three arrays `xPrimLR`, `yPrimLR` and `zPrimLR` six arrays `xPrimL`, `xPrimR`, ..., `zPrimR` are needed. Therefore, preprocessor macros have been introduced to switch from one representation to the other. Variable declarations for the ADCL objects were added.

```
integer  :: adcl_vmap          ! ADCL vector map object
integer  :: adcl_topo          ! ADCL topology

! ADCL vector and request objects for the primitive variables
integer  :: adcl_vec_pvar, adcl_request_pvar

! ADCL vector and request objects of the left and right states
! for the Riemann problem
integer  :: adcl_vec_xpriml, adcl_request_xpriml
integer  :: adcl_vec_xprimr, adcl_request_xprimr
integer  :: adcl_vec_ypriml, adcl_request_ypriml
integer  :: adcl_vec_yprimr, adcl_request_yprimr
integer  :: adcl_vec_zpriml, adcl_request_zpriml
integer  :: adcl_vec_zprimr, adcl_request_zprimr

include 'ADCL.inc'            ! ADCL internal variables
```

These ADCL objects are initialized and freed in two new routines `iniADCL` and `closeADCL` shown in Fig. 5.2. Instead of the original MPI implementation, calls to `ADCL_Request_start` are executed:

```
call ADCL_Request_start( adcl_request_pvar, MPI%ierror)
```

and

```
call ADCL_Request_start( adcl_request_xPrimL, MPI%ierror )
call ADCL_Request_start( adcl_request_xPrimR, MPI%ierror )
call ADCL_Request_start( adcl_request_yPrimL, MPI%ierror )
call ADCL_Request_start( adcl_request_yPrimR, MPI%ierror )
call ADCL_Request_start( adcl_request_zPrimL, MPI%ierror )
call ADCL_Request_start( adcl_request_zPrimR, MPI%ierror )
```

respectively. Figure 5.3 shows the changes to the main program. Attention should be paid to the fact that with only these few code modifications the performance of the MPI communications is now portable thanks to the use of ADCL.

5.3.2 Setup

Performance data is generated using Euler3D. Since the neighborhood communication is implemented using ADCL, the library will evaluate, given a brute-force search, all available codelets for this communication pattern a given

```
SUBROUTINE iniADCL(pvar, LRARGS, CONST, MESH, MPI)
  integer :: nhalolayers = 1   ! number of halo layers
  integer :: ndim = 3          ! number of dimensions
  integer :: dims(ndim)        ! extent of dimensions
  ! ....

  call ADCL_Init ( MPI%ierror )
  call ADCL_Topology_create ( MPI%comm_cart, adcl_topo, MPI%ierror )

  call ADCL_Vmap_halo_allocate( 1, adcl_vmap, MPI%ierror )

  dims(1) = MESH%IMAX+2;  dims(2) = MESH%JMAX+2;  dims(3) = MESH%KMAX+2

  call ADCL_Vector_register_generic ( 3,  dims, NVAR, &
       MPI_DOUBLE_PRECISION, pvar, adcl_vec_pvar, MPI%ierror )
  call ADCL_Request_create ( adcl_vec_pvar, adcl_vmap, adcl_topo, &
       ADCL_FNCTSET_NEIGHBORHOOD, adcl_request_pvar, MPI%ierror)

  call ADCL_Vector_register_generic ( 3,  dims, NVAR, &
       MPI_DOUBLE_PRECISION, xpriml, adcl_vec_xpriml, MPI%ierror )
  call ADCL_Request_create ( adcl_vec_xpriml, adcl_vmap, adcl_topo, &
       ADCL_FNCTSET_NEIGHBORHOOD, adcl_request_xpriml, MPI%ierror)
  ... ! same for xprimr, ypriml, yprimr, zpriml, zprimr
END SUBROUTINE iniADCL

SUBROUTINE closeADCL(MPI)
  call ADCL_Topology_free ( adcl_topo, MPI%ierror )

  call ADCL_Request_free ( adcl_request_pvar, MPI%ierror )

  call ADCL_Vector_deregister ( adcl_vec_pvar, MPI%ierror )
  call ADCL_Vector_deregister ( adcl_vec_xpriml, MPI%ierror )
  ... ! same for xprimr, ypriml, yprimr, zpriml, zprimr

  call ADCL_Vmap_free ( adcl_vmap, MPI%ierror )

  call ADCL_Finalize ( MPI%ierror )
END SUBROUTINE closeADCL
```

Figure 5.2: ADCL implementation of Euler3D: initialization and deallocation of ADCL objects

```
PROGRAM Euler3D
! Module inclusions and variable declarations
  ! Initialize Simulation Parameters
  ! Initialize Mesh
#ifdef ADCL
    CALL iniADCL(pvar, LRARGS, CONST, MESH, MPI)
#endif
  ! Set Initial Condition
  ! Initialize exact solution module for problems with known
  !   exact solutions

  ! Main program loop over time steps

#ifdef ADCL
  call closeADCL(MPI)
#endif
  ! Output & Finalize
END PROGRAM Euler3D
```

Figure 5.3: ADCL implementation of Euler3D: modified main program.

number of times during the initial iterations of the solver. We use the twelve different codelets for the neighborhood communication which correspond to the MPI 1 standard (cf. table 3.1). Using the statistical methods presented, an estimated average execution time is calculated for every codelet and the codelet with the smallest one is selected.

The runs generating input for the analysis using a brute-force search as well as the verification runs have been executed three times using the MPI library Open MPI v1.4. During each run using the runtime selection logic, 100 data points have been generated per codelet and process. The prediction quality of the four evaluation methods was analyzed. During each verification run (cf. sec. 3.3.2), the execution times of 500 time steps were stored and subsequently averaged over all three runs. In order to make the conditions as comparable as possible, the reference data was produced within the same batch scheduler allocation and thus had the same node assignments.

5.3.3 Results

The communication time of an ADCL run consists mainly of the communication during the search phase where also non-optimal codelets are tested and the performance of the codelet selected by the decision algorithm.

After defining what we understand by "best" or "worst-performing" codelet in sec. 5.3.3.1, attention is paid to two questions. The first question is of more general nature: what maximum performance gains resp. losses could we expect when using ADCL instead of the original MPI implementation. This depends on the calling overhead of ADCL and the quality of the implemented codelets. In the best case, we know the best-performing codelet beforehand and avoid the search and its overhead completely. This is exactly the verification run with the smallest execution time. In the worst case, we select the slowest codelet. All codelets during the search phase will be faster than the slowest codelet. In consequence, the verification run with the largest execution time constitutes an upper bound. So in sec. 5.3.3.2, the data from the verification runs will be compared to the original MPI implementation to establish potential gains and losses and it will be investigated which codelets do actually perform well in the different settings.

The second question is what is actually gained. This depends on the quality of the statistical method used in the decision algorithm, i.e. how many data points are needed during the search and how well the selected codelet performs. It also depends on the performance of each codelet which is tested during the search phase, but this overhead can only be influenced indirectly by shortening the length of the search. So in sec. 5.3.3.3, we will evaluate the choices made by the statistical methods, investigate what performance penalties arise if the choice was not correct and describe the influence of amount of measurements on the selection. Finally, sec. 5.3.3.4 states, what performance gains or losses one would have obtained for the test case using ADCL with the heuristic based on a 100 data points per codelet.

5.3.3.1 Characterizing the performance of codelets

Measurements always introduce a certain amount of variability. Causes for deviations when measuring communications times can be network flooding resulting from the communication algorithm, a poor implementation of parts of the MPI library or shared resources as well as sometimes hardware configuration challenges based on frequency scaling of the processor or the simultaneous multi-threading leading to different process arrival patterns. These problems occur without influence of the user and there is no use to do one's utmost to circumvent them.

Figure 5.4 shows an example where there is no single best codelet. It shows the average execution times from three verification runs along with their minimum and maximum values as error bars for different codelets. Any of the codelets c_6, c_0 or c_7 might be the fastest, depending on its performance relative to the others. On the other hand, c_7 could also be the slowest codelet or any of the codelets

c_1, c_3 and c_5. We will encounter this problem again in a different context in chapter 6.

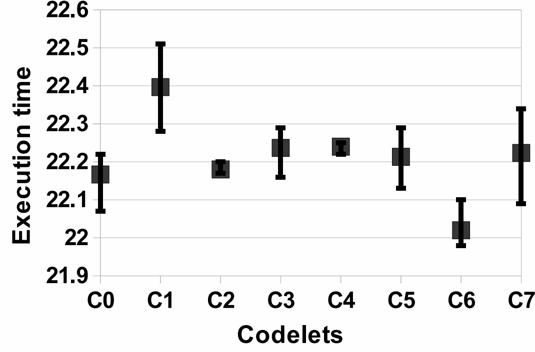


Figure 5.4: Example of multiple best or worst-performing codelets. The figure displays the average execution time $t_v^{\mathrm{avg}}(c_i)$ of 3 verification runs. Y error bars show the range of average execution times.

Let $t_v^i(c), i = 1, \dots, n_r$ denote the verification run times of a codelet c and

$$t_v^{\mathrm{avg}}(c) = \frac{1}{n_r} \sum_{i=1}^{n_r} t_v^i(c), \tag{5.8}$$

$$t_v^{\min}(c) = \min_{i=1,\dots,n_r} t_v^i(c) \text{ and} \tag{5.9}$$

$$t_v^{\max}(c) = \max_{i=1,\dots,n_r} t_v^i(c) \tag{5.10}$$

the average, minimum and maximum verification run time of the codelet.

In the following, operations over i like $\min_i$ abbreviate $\min_{i=1,n_r}$ and over a codelet c like $\sum_c$ stands for the sum over all codelets in the set of codelets. We define the overlap $o(c, d)$ for a codelet c with a codelet d as

$$o(c,d) = o(d,c) = \begin{cases} 0 & \text{if } t_v^{\min}(c) > t_v^{\max}(d) \text{ or } t_v^{\max}(c) < t_v^{\min}(d) \\ 1 & \text{else} \end{cases} . \tag{5.11}$$

The overlap is 0 if the codelets can not switch rankings and 1 otherwise. So the *best-performing* codelets or *winner codelets* are the ones with the minimum verification run time along with those which have an overlap of 1 with it. The *worst-performing codelets* are those with the maximum verification run time along with those which have an overlap of 1 with it.

5.3.3.2 Potential Performance Benefits of ADCL

In this section, we will analyze ADCL's potential to increase application performance if the best-performing codelet is selected.

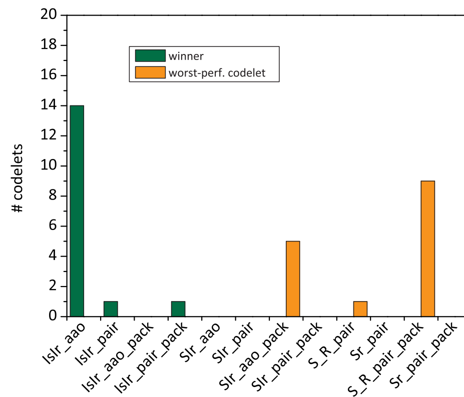
Best- and Worst-performing Codelets Table 5.2 shows the best- resp. worst-performing codelets for different settings. On *Laki* with InfiniBand, best- and worst-performing codelets are clearly defined except for the large test case on 2048 processors. `IsIr_aao` is the codelet of choice across different numbers of processes and for different problem sizes. As InfiniBand networks hardly get congested, a non-blocking communication is well-suited for this type of network. This assessment is confirmed when looking at the worst-performing codelets, which are often based on Send/Recv.

On *shark*, the variations of the execution times for the verification runs are higher, leading to multiple best- and worst-performing codelets in the different settings. In contrast to *Laki*, the problem size or the number of processes do influence the outcome. `IsIr_aao` is also often among the winners, as the amounts of data which are communicated are moderate and do not saturate the Ethernet interconnect. Blocking communications perform partially well like `SendRecv`, but codelets based on Send/Recv are frequently among the worst-performing ones. That `S_R_pair_pack` does not perform well across interconnects indicates that either this implementation is ill-suited for the problem or that the implementation itself can be improved.

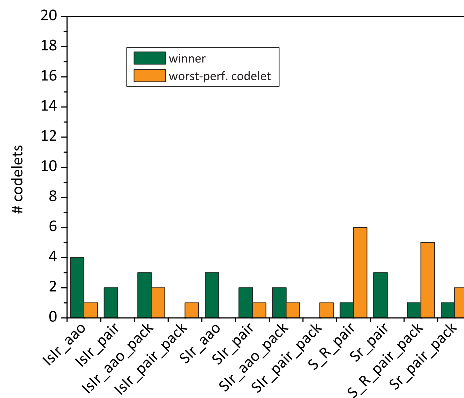
Figure 5.5 shows the result if one calculates the sum of all the cases in which a codelet performs best or worst on *Laki* with InfiniBand, on *shark* with Ethernet and in total. One can see, that for *Laki* with InfiniBand, winners and worst-performing codelets are clearly defined whereas on *shark* with Ethernet, codelets which perform best in one setting do perform worst in another and vice versa. In total, 11 out of 12 codelets do perform best in some setting, and 9 out of 12 codelets count to the worst-performing in some setting. The codelet based on `MPI_Isend/MPI_Irecv/MPI_Waitall` performs best for most settings, but also twice moderate and worst once. Only a change of problem size or number of MPI processes lead to this extremely different behavior. This shows again the importance of an automatic empirical optimization library. Otherwise, optimizations might have lead to an implementation of `IsIr_aao` which would entail performance losses in 3 of 35, i.e. about 9% of the settings. With ADCL, the choice of the codelet is not fixed, so that the library can adapt itself to the current settings.

setting	IsIr_aao	IsIr_pair	IsIr_aao_pack	IsIr_pair_pack	SIr_aao	SIr_pair	SIr_aao_pack	SIr_pair_pack	S_R_pair	Sr_pair	S_R_pair_pack	Sr_pair_pack
laki_32S_ib	w										l	
laki_64S_ib	w										l	
laki_128S_ib	w								l		l	
laki_256S_ib	w										l	
laki_512S_ib	w						l					
laki_1024S_ib	w						l					
laki_2048S_ib	w						l					
laki_32L_ib	w										l	
laki_64L_ib	w										l	
laki_128L_ib	w										l	
laki_256L_ib	w										l	
laki_512L_ib	w										l	
laki_1024L_ib	w						l					
laki_2048L_ib	w	w		w			l					
shark_8XS_eth	l	w	l		w	l			l	w	l	l
shark_16XS_eth			w				w		l		l	w
shark_32XS_eth			l	l			l	l	w		w	l
shark_48XS_eth	w	w				w			l	w	l	
shark_8S_eth	w		w						l		l	
shark_16S_eth	w				w				l		l	
shark_32S_eth	w		w		w	w	w		l	w		

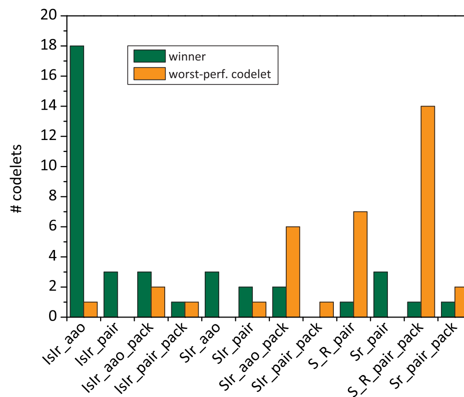
Table 5.2: Overview of best-performing (w) and worst-performing (l) codelets.



(a) on *Laki* with InfiniBand



(b) on *shark* with Ethernet



(c) total

Figure 5.5: Sum of cases in which codelet performed best or worst.

Maximum Performance Gains and Losses We will now demonstrate that the application performance can benefit of the integration of ADCL. We compare the runtime of the original code (denoted as `noadcl`) with the those of the verification runs for the best- and worst-performing codelet. Also results for the modified code with multiple arrays for the state variables (denoted as `noadcl-lr`) are given for comparison.

In the following, the subscript w refers to the winner codelet, l to the worst-performing codelet, o to the original implementation without ADCL and m to the modified Euler3D code with multiple arrays. Let $t_k^i, i = 1, \ldots, n_r,\ k \in \{o, m, w, l\}$ denote the run times of the four cases and

$$\begin{aligned} t_k^{\text{avg}} &= 1/n_r \sum_{i=1}^{n_r} t_k^i, \\ t_k^{\text{min}} &= \min_{i=1,\ldots,n_r} t_k^i \text{ and} \\ t_k^{\text{max}} &= \max_{i=1,\ldots,n_r} t_k^i \end{aligned}$$

the average, minimum and maximum run times.

The average execution times $t_k^{\text{avg}},\ k \in \{o, m, w, l\}$ for the different code versions of the $n_r = 3$ verification runs are depicted on the left hand side of fig. 5.6. The error bars are simply $\left[t_k^{\text{min}} - t_k^{\text{avg}}, t_k^{\text{max}} - t_k^{\text{avg}}\right]$. The changes introduced in Euler3D to integrate ADCL did not turn out favorably. However, it does not influence the performance of ADCL, as its data transfer primitives are based on derived data types or pack/unpack, and not on buffers.

The possible percental gains resp. losses of ADCL shown in fig. 5.6 on the right hand side, are given by

$$g_{\text{avg}}(k) = \frac{t_k^{\text{avg}} - t_o^{\text{avg}}}{t_o^{\text{avg}}} \cdot 100\%,\ k \in \{w, l\}$$

The error bars yield for the winner codelet

$$\left[g_{\text{avg}}(w) - \frac{t_w^{\text{min}} - t_o^{\text{max}}}{t_o^{\text{max}}} \cdot 100\%,\ \max\left(0, \frac{t_w^{\text{max}} - t_o^{\text{min}}}{t_o^{\text{min}}} \cdot 100\% - t_w^{\text{avg}}\right)\right]$$

and for worst-performing codelet

$$\left[\max\left(0, g_{\text{avg}}(l) - \frac{t_l^{\text{min}} - t_o^{\text{max}}}{t_o^{\text{max}}} \cdot 100\%\right), \frac{t_l^{\text{max}} - t_o^{\text{min}}}{t_o^{\text{min}}} \cdot 100\% - g_{\text{avg}}(w)\right]$$

If multiple winners or worst-performing codelets are present, t_w^{min} and t_w^{max} as well as t_l^{min} and t_l^{max} are global minima resp. maxima of all those codelets.

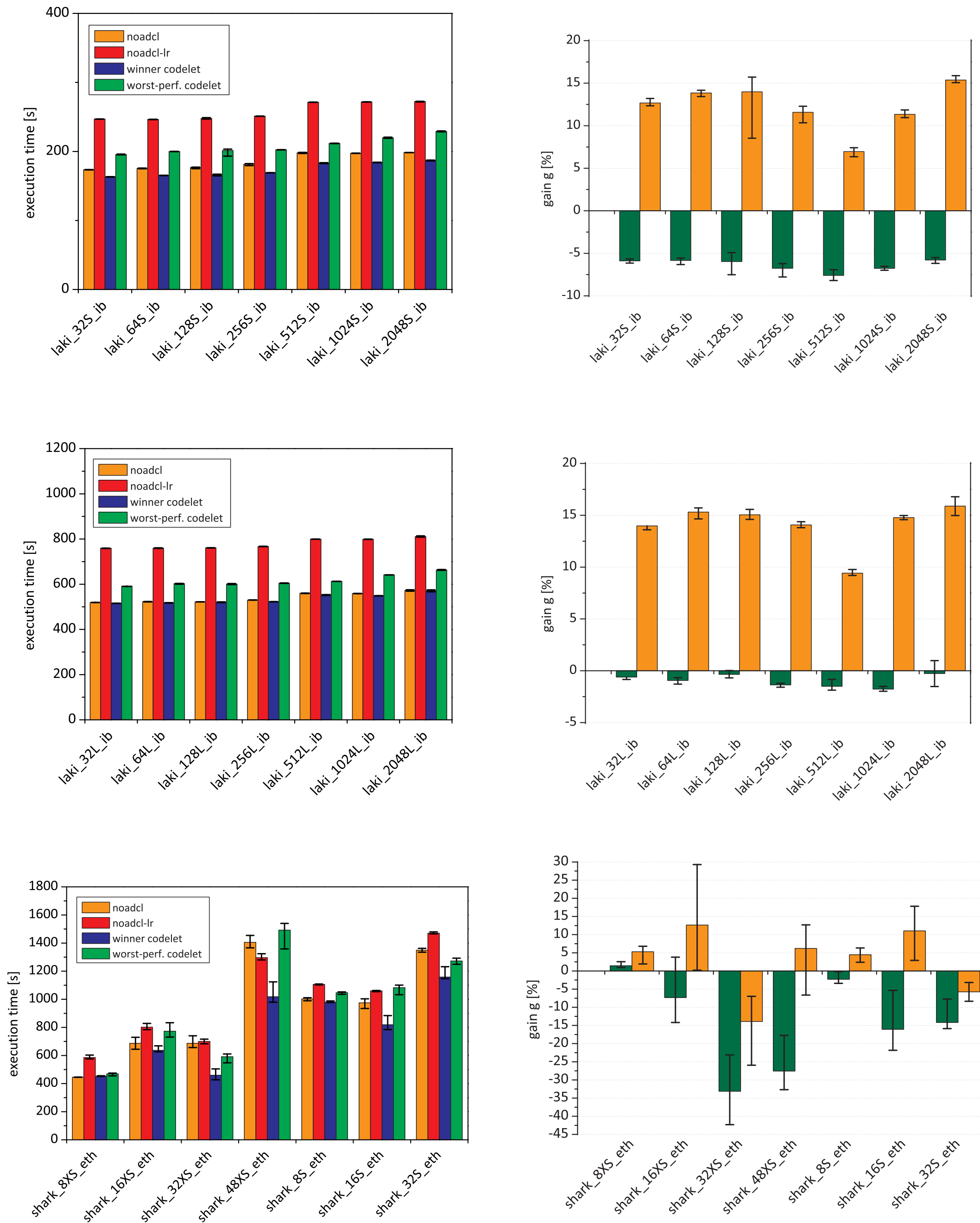


Figure 5.6: Execution times of original code (noadcl), the modified code with multiple arrays for the state variables (noadcl-lr) compared to the execution times using the best-performing and the worst-performing codelet (left side). Percental gains of best codelet vs. potential losses of worst codelet (right side).

Laki with InfiniBand shows some potential for optimization for small test cases and hardly any for the large test case. For *shark*, performance gains might reach up to over 30%. For the XS problem size on 8 processors, there will always be slight performance losses of $1-5\%$ when using ADCL while on 32 processors, ADCL always leads to performance gains in the range of $14-33\%$. On the other hand, the graphics show that using ADCL might also decrease the overall performance if the decision logic does not work properly.

5.3.3.3 Statistical Data Analysis

Measurements on *Laki* with InfiniBand The first set of tests have been executed on *Laki*. This work first evaluates the statistical approaches using ADCL performance data from runs over the InfiniBand network with $32, 128, \ldots, 2048$ MPI processes with problem sizes S and L (cf. tbl. 2.1). As described in the introduction of this chapter, the InfiniBand network produces highly reliable performance data with only few outliers. So many of the decisions without data filtering can be regarded as reasonable and the heuristic shows nearly exactly the same behavior as without data filtering.

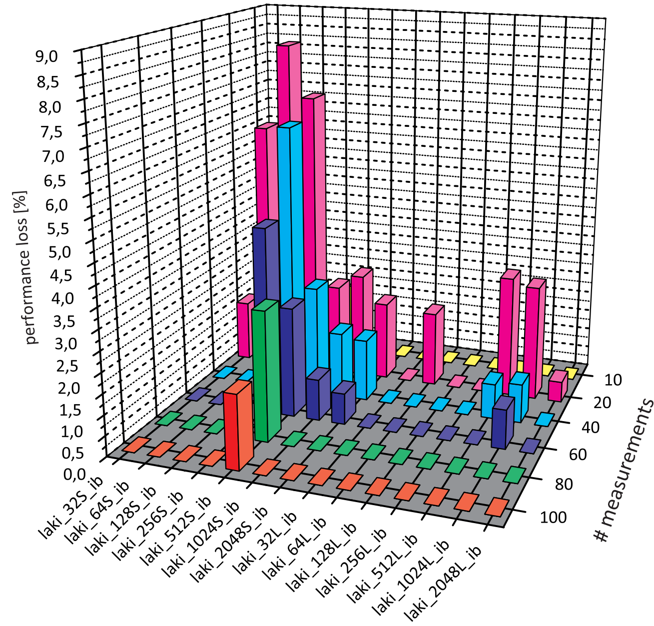
With $n_m = 100$, all statistical methods selected in all cases except for the 512S setting correctly codelet c_0 as winner. In this case, the selected codelet performs $2-3\%$ worse than the winner codelet as is shown in tbl. 5.3.

	winner codelet			avg. overhead [%]		
	run1	run2	run3	run1	run2	run3
no filtering	1	1	1	1.8	1.8	1.8
heuristic	1(f)	1(f)	1 (f)	1.8	1.8	1.8
IQR	0	0	0	0	0	0
cluster analysis	9	1	9	3.2	1.8	3.2
robust statistics	1	0	0	1.8	0	0

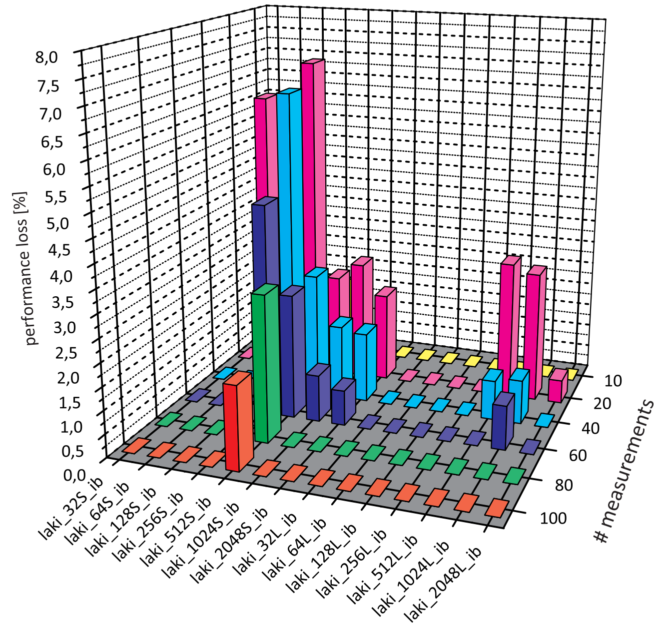
Table 5.3: Testcase 512S_ib with incorrect codelets judged best-performing.

If one reduces n_m from 100 to $80, 60, 40, 20$ or 10 as depicted in fig. 5.7, using no filtering or the heuristic leads to disadvantageous decisions in some cases, whereas the method based on IQR, cluster analysis or robust statistics are nearly insusceptible to variations of n_m. Please notice the different scales for figs. 5.7(a)–(b) and figs. 5.7(c)–(e).

To summarize the results of the InfiniBand tests, all four statistical approaches presented in the previous section performed reasonably well. Whereas the heuristic needed 80–100 data points per codelet, the other methods' choice were in most cases correct with as little as 10 measurements per codelet.

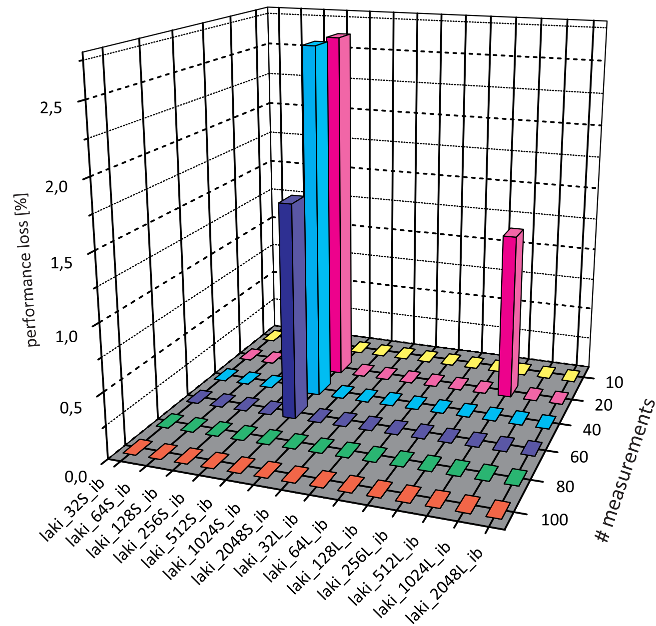


(a) no filtering

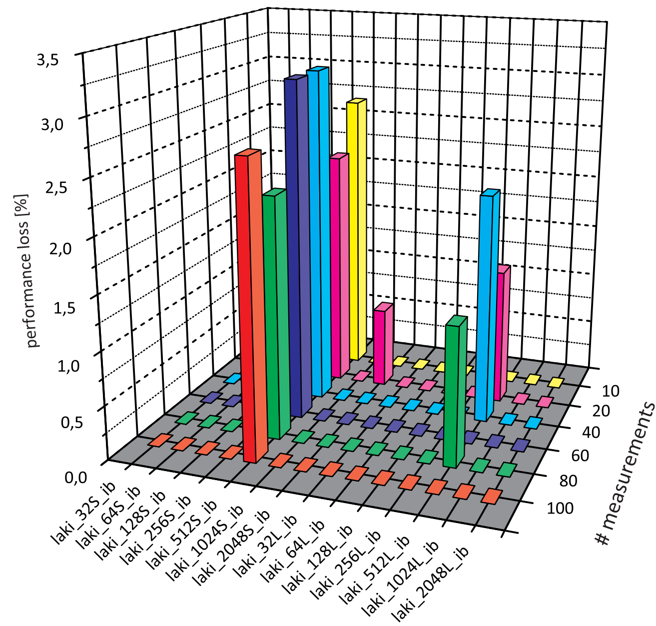


(b) heuristic

Figure 5.7: Overheads of the selected codelet compared to the winner codelet on *Laki* with InfiniBand for different settings and different numbers of measurements n_m per codelet.

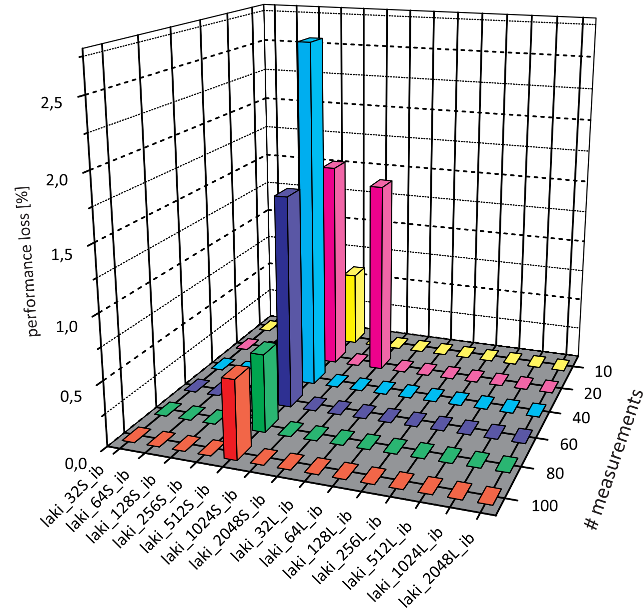


(c) IQR



(d) cluster analysis

Figure 5.7: Overheads of the selected codelet compared to the winner codelet on *Laki* with InfiniBand for different settings and different numbers of measurements n_m per codelet.

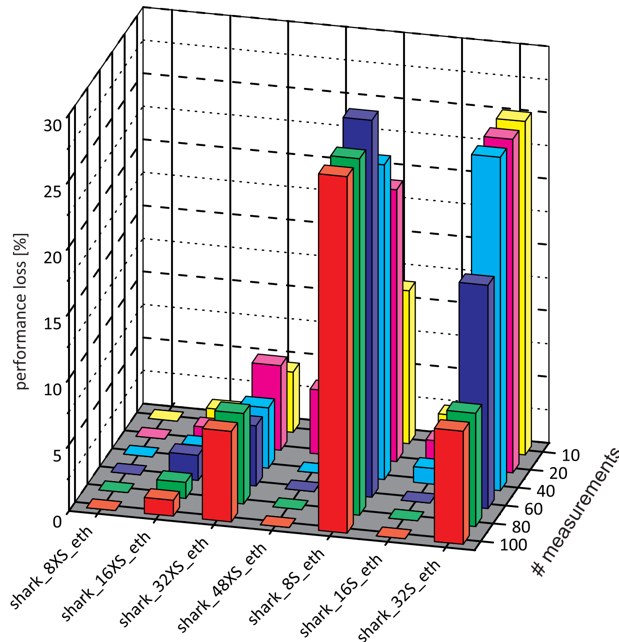


(e) robust statistics

Figure 5.7: Overheads of the selected codelet compared to the winner codelet on *Laki* with InfiniBand for different settings and different numbers of measurements n_m per codelet.

Measurements on *Shark* with Gigabit Ethernet The same tests are repeated with 8, 16, 32 and 48 processes for the XS problem size and 8, 16 and 32 processes for the small problem size using the Gigabit Ethernet interconnect on *shark*. The Gigabit Ethernet switch used within this network provides a full duplex 1GBit connection for each node. All tests were executed in virtual node mode, so that two MPI processes are running on each node and have to share one physical link. This increases the number of outliers, but it would be unreasonable to leave half of the processors idle if no memory or performance issues militate against it.

All statistical methods share the behavior of the approach without outlier-filtering (cf. figs. 5.8(a)–(e)). In the settings 32XS, 8S and 32S, unfavorable codelets have been selected leading to overheads of about 5%, 25% and 10% compared to the winner codelet. The influence of n_m is negligible.

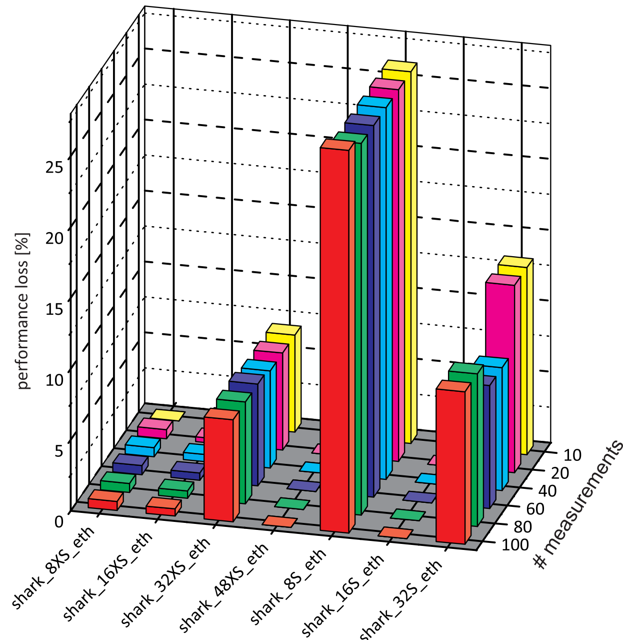


(a) no filtering

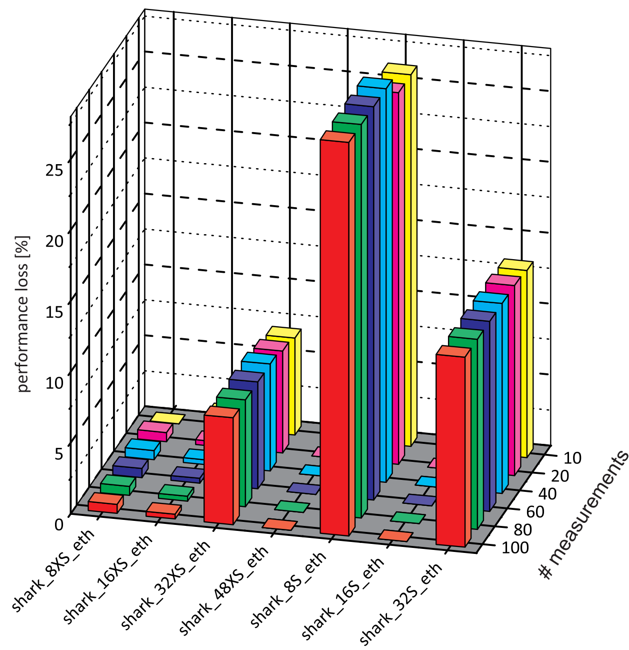
Figure 5.8: Overheads of the selected codelet compared to the winner codelet on *shark* with Ethernet for different settings and different numbers of measurements n_m per codelet.

Looking closer into the performance data, one realizes that the wrong decisions can not be attributed to the statistical methods, but are inherent in the data. Figure 5.9 shows the performance data of the different codelets for one setting. Most communications take around 10 ms with a few outliers in the 200 ms range. The red lines depict the average execution times of the verification runs for each codelet. In general, one can extrapolate from the data points to the performance of the codelet in the production phase, but for `IsIr_aao` and `SIr_aao` the data points suggest a better performance than actually achieved. This is most likely due to the influence of `MPI_barrier` which is called during the search phase before the measurement is started to avoid that subsequent communications influence one another.

To summarize the results achieved over the Gigabit Ethernet network on *shark*, the quality of the predictions for all statistical methods is comparable and strongly depends on the reliability of the underlying data. The number of measurements has hardly any influence on the decisions taken.

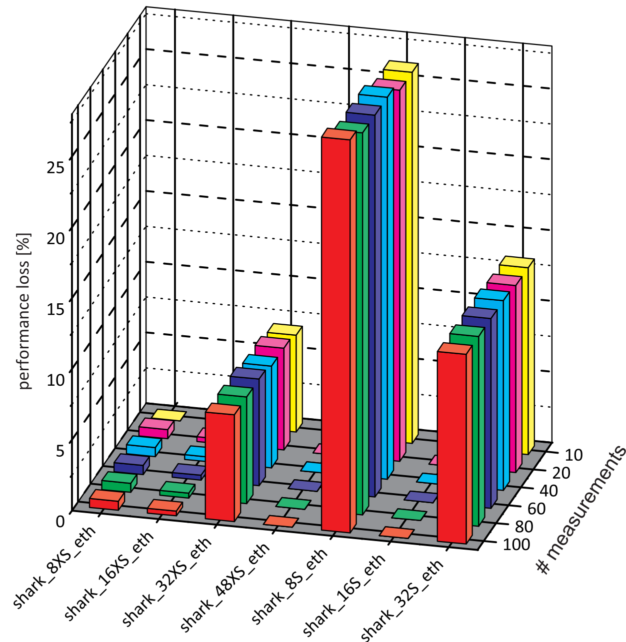


(b) heuristic

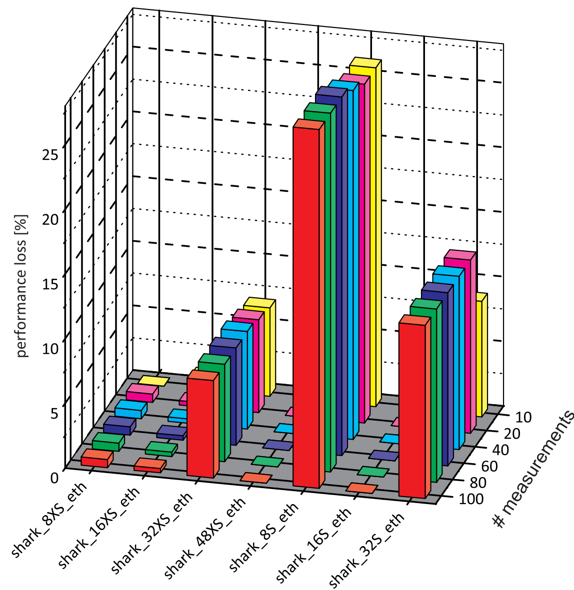


(c) IQR

Figure 5.8: Overheads of the selected codelet compared to the winner codelet on *shark* with Ethernet for different settings and different numbers of measurements n_m per codelet.



(d) cluster analysis



(e) robust statistics

Figure 5.8: Overheads of the selected codelet compared to the winner codelet on *shark* with Ethernet for different settings and different numbers of measurements n_m per codelet.

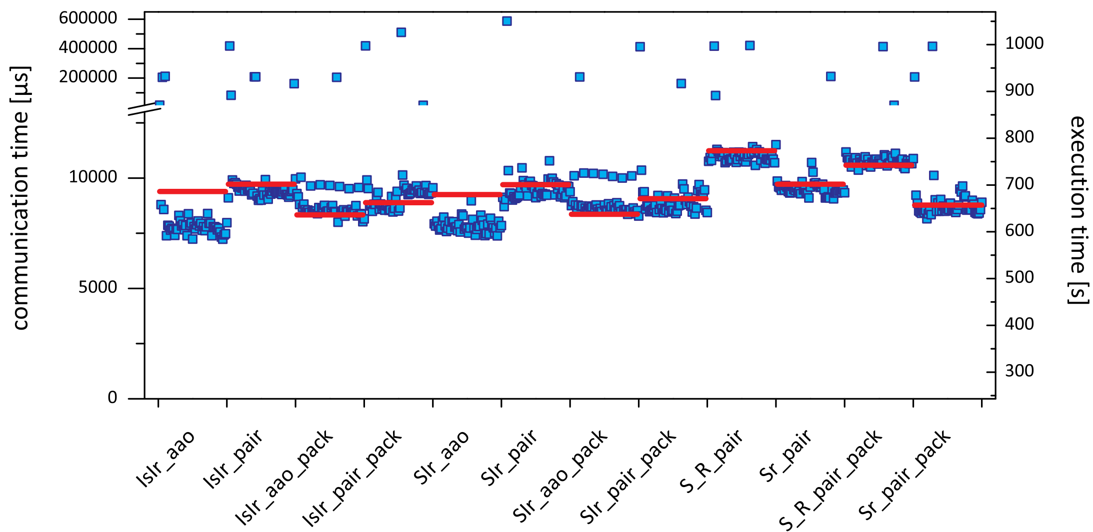


Figure 5.9: Measurements that do not reflect the performance of the verification runs (red lines).

5.3.3.4 Performance of ADCL

Table 5.4 shows that even with non-optimal choices of the decision logic of ADCL, using ADCL is still worth-while. For the small problem size on *Laki* with InfiniBand, performance gains for the Gauss pulse after 500 time steps were around 5%. On *shark* with Ethernet interconnect, performance gains reached 17%. In any setting, the use of ADCL did not result in performance degradations of more than 2%.

5.4 Conclusion

This chapter discussed four different approaches for handling outliers in parallel performance measurements, namely a heuristic derived from the trimmed mean value, a standard approach using interquartile ranges, an approach based on cluster analysis and finally a method using robust statistics. Although using fundamentally different approaches, all methods in our evaluation have shown the capability to handle outliers in most scenarios. The heuristic tended to need more data points for a correct decision than the other methods. However, even with a non-optimal selection, ADCL could improve the application's performance in some cases or did at least not degrade it.

setting	avg. gain	Yerr−	Yerr+
laki_32S_ib	-4.86	0.29	0.43
laki_64S_ib	-4.64	0.56	0.5
laki_128S_ib	-4.39	1.74	1.34
laki_256S_ib	-5.25	1.04	0.57
laki_512S_ib	-4.5	0.42	0.34
laki_1024S_ib	-4.95	0.24	0.41
laki_2048S_ib	-4.12	0.32	0.39
laki_32L_ib	-0.32	0.19	0.27
laki_64L_ib	-0.17	0.32	0.35
laki_128L_ib	0.83	0.56	0.84
laki_256L_ib	-0.38	0.18	0.15
laki_512L_ib	-0.65	0.81	0.9
laki_1024L_ib	-0.36	0.25	0.44
laki_2048L_ib	0.48	0.7	0.76
shark_8XS_eth	1.3	0.6	0.56
shark_16XS_eth	1.46	7.22	9.33
shark_32XS_eth	-12.07	6.89	4.42
shark_48XS_eth	-17.05	5.41	4.11
shark_8S_eth	-1.69	1.01	1.31
shark_16S_eth	-11.7	5.37	9.82
shark_32S_eth	-7.39	1.97	1.86

Table 5.4: Performance gains for the test case after 500 time steps when using ADCL with heuristic and 100 data points per codelet.

The major advantage of the interquartile range method and of the heuristic approach is the low computational complexity compared to cluster analysis and robust statistics. This is especially important for a library which has to perform the operations at application runtime.

In [102], a similar investigation was carried out using a TFQMR solver on *Laki* with InfiniBand, the *cacau* cluster with a hierarchical Gigabit Ethernet, which is now out of service, and on *shark* with InfiniBand and Ethernet interconnects. It was shown that the approach applying techniques from robust statistics demonstrates a superior behavior when the performance data has a low variance, and the main task is to determine subtle differences between the implementations. On the other hand, the same approach had significant problems when the data was widely distributed in multiple, large data clusters. The approach using cluster analysis did have similar problems in the latter scenario, mainly due to the removal of too many clusters. The standard approach using interquartile range showed a fair behavior for any of the analyzed scenarios. The heuristic based on the trimmed mean value did show the most reliable performance, delivering the optimal or close to optimal decisions in most of the analyzed scenarios.

One important result of this chapter is that the method how the performance data was collected in the past, is not producing reliable data in some cases. This problem lead to the development of the new data collection methods presented in chapter 6.

6 Evaluation of Timing Techniques to Generate Empirical Data for Collective Communications

Motivated by the problems encountered in the last chapter, this chapter investigates how to assess the performance of different codelets implementing collective MPI communications in an empirical optimization library. The generation of empirical data is an essential and critical part of the library as the data constitutes the basis for the optimization process to select the best-performing codelet.

Four different timing techniques to collect data are presented in sec. 6.1. A *timing technique* is a certain way of how to measure communication times to generate empirical data. Two of them are based on a newly introduced ADCL object, the ADCL timer object, which allows not only to time a codelet itself, but also (part of) its environment. Its technical realization is explained in sec. 6.2. This work evaluates the timing techniques and the impact of the timer object to generate accurate empirical data taking the optimization of the collective in the FFT NAS Parallel Benchmark as example. The results on a variety of systems with different MPI implementations are presented in sec. 6.3.

6.1 Timing Techniques to Generate Performance Data

A timing technique which generates data for the empirical optimization process has to match different, competing goals. Firstly, it is favorable if the timing technique creates no significant additional overhead during the search phase, i.e. that no additional synchronization for the timing is needed. Secondly, to keep the search phase—and thus its overhead—short, the timing technique should generate reliable data with few outliers, so that a small number of measurements suffices to decide on the best-performing codelet. Thirdly and most importantly, the data has to reproduce the behavior observed in long-term runs, i.e. that despite the execution of additional code of the timing

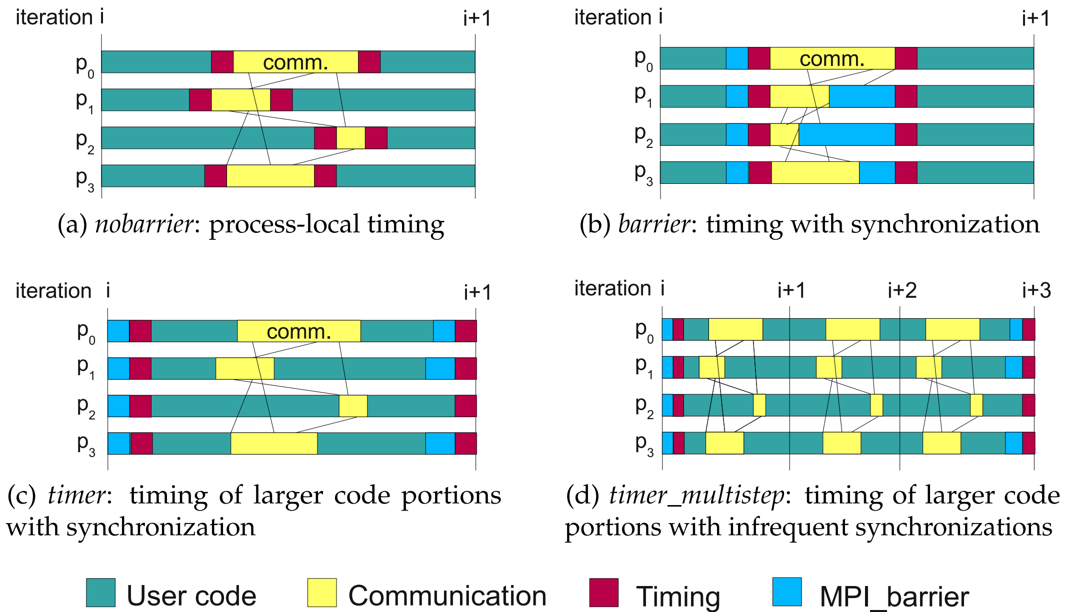
technique, data comparable to the original execution behavior of the program without timing is produced. Otherwise, a wrong codelet with a non-optimal performance is chosen and the overall performance degrades.

Measuring collective communication times is more difficult compared to a point-to-point communication. Per se, each process only knows its own communication time, but it is impossible to state the total execution time of the communication without additional communication due to the asynchronous nature of a parallel program. In the rare case that synchronized clocks are available, their use would lead to an additional collective of type `allreduce` to compute the maximum over all local communication times. Otherwise synchronization has to be enforced manually by calls to `MPI_Barrier` to get an estimate of the total execution time of the collective.

The first and simplest timing technique called *nobarrier* is to embrace the collective communication with timing routines as depicted in fig. 6.1a and use after a post-processing step the maximum local execution time as an estimator for the total execution time of the collective communication. This technique causes practically no additional overhead and is transparent to the user, as it can be hidden inside ADCL, but has a number of drawbacks. Apparently, it may fail as there is no correlation between local execution time and its effect on the total execution time as, e.g., frequent delays of a short communication phase on one process can cause an increase in the total execution time. As processes are not synchronized, the current communication may influence the subsequent one. This may result in wrong preconditions when switching between codelets. In addition, if execution times are very short, system timers eventually lack precision.

A solution which measures local execution times and still captures the total execution time is to add synchronizations before the calls to the timing routines. This constitutes the second timing technique *barrier* shown in fig 6.1b. The synchronizations create additional overhead during the search phase and the measurements represent a worst-case scenario since eventually delays would cancel. Also systematic errors may be introduced which falsify the empirical data since application features such as process arrival patterns are annulated before the measurement is started and processes do not leave the barrier at the exact same time. As for *nobarrier*, the timing technique is transparent for the user, but system timers might lack precision.

To mimic the application behavior, this work introduces a new ADCL object, the timer object, which is detailed in sec. 6.2. The timer object does not just measure the codelet in isolation, but rather the codelet plus (a part of) its environment, e.g. one or more iterations, as shown in fig. 6.1c and (d). New interfaces offer handles to the timing routines so that the region which is measured is now



(a) *nobarrier*: process-local timing

(b) *barrier*: timing with synchronization

(c) *timer*: timing of larger code portions with synchronization

(d) *timer_multistep*: timing of larger code portions with infrequent synchronizations

Figure 6.1: Different timing techniques to generate empirical data for collective communication operations.

controlled by the user and no longer by the library. Its obvious purpose is to imitate the original execution behavior of the program and avoid distortions caused by synchronizations directly before and after the execution of the codelet such as for the *barrier* technique. As execution times are considerably larger, problems with the accuracy of machine timers are unlikely. However, the user has now the choice but also the responsibility to select the right code portion to measure.

Timing techniques based on the timer object use synchronization for their measurements to make sure that subsequent measurements do not influence one another. Let $n_s \geq 1$ denote the number of iterations between two synchronizations and $m, m = 1, \ldots$ the m-th measurement. Then the MPI processes are synchronized during the $(1 + (m-1) \cdot n_s)$-th iteration just before the timing is started and during the $m \cdot n_s$ iteration just before the timing is stopped. For the timing technique *timer* with the synchronization interval $n_s = 1$ the treatment of outliers as explained in chapter 5 does not change. For *timer_multistep*, n_s is set to a value greater than 1, but the number of executions of a specific codelet during the search phase is left the same, i.e. $m * n_s = const.$ to keep the search effort the same. This results in fewer already averaged measurements per codelet, which might be more susceptible to the outlier problem. With well-balanced values for m and n_s, this technique should demonstrate an "undisturbed" execution

behavior but also provide enough measurements to allow outlier handling.

6.2 Technical Concept of the ADCL Timer Object

In the following, the design of the timer object is explained in detail by looking deeper into ADCL internals. Its goals are to

1. allow to time larger code portions *(i)* to avoid distortions caused by synchronizations directly before or after the execution of the codelet and therefore enhance the quality of the empirical data as well as *(ii)* to enable the optimization of communication operations that are part of larger code portions, such as code sections which overlap communication and computation,
2. allow the simultaneous optimization of different communications, i.e. multiple requests, which may interfere with one another and to
3. agree well with the existing systematic of ADCL objects to allow maximum code reuse.

The first goal sets up requirements of how the measurement of the execution times of a single request should be carried out. Taken by itself, it would already allow to compare the different timing techniques without having to introduce a new timer object. The second goal, however, requires a timer object as multiple requests need to be controlled simultaneously. We will first explain the code changes necessary to achieve the first goal and then the concept of the timer object leading to the second goal.

6.2.1 Case 1: Optimizing One Request

The timing methods *nobarrier* and *barrier* already existed in ADCL. Their pseudocode is shown in figs. 6.2(a) and (b). Grey portions indicate user code, white portions are parts inside ADCL. A call to `ADCL_Request_start` times and executes the codelet. To fulfill the goal mentioned, two additional interfaces, `ADCL_Timer_start` and `ADCL_Timer_stop` to start and stop the timing are provided. This leaves only the task to execute the codelet to `ADCL_Request_start`. This results in the timing techniques *timer* and *timer_multistep* as shown in figs. 6.2(c) and (d). One can clearly see that *nobarrier* and *barrier* are more userfriendly as *timer* and *timer_multistep* since they require one single call to the library instead of three. The latter two, however, offer the possibility to insert additional code or other communications before and after the execution of the codelet which are included in the measurement.

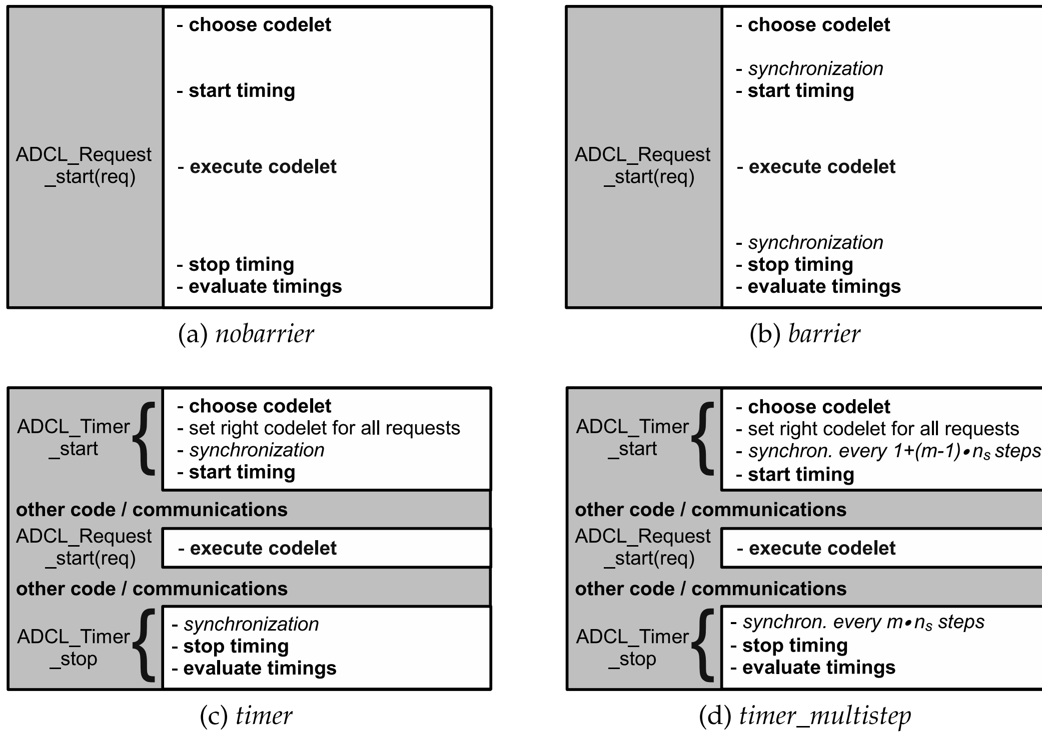


Figure 6.2: Schematic view of implementation of timing techniques with user code (grey) and ADCL code (white). For the timing techniques *nobarrier* and *barrier* (figs. (a) and (b)), the timing routines embrace only the codelet whereas for *timer* and *timer_multistep* (figs. (c) and (d)), they include the codelet plus its environment where m denotes the m-th measurement resp. call to start or stop the timing and $n_s \geq 1$ the synchronization interval.

6.2.2 Case 2: Optimizing Multiple Requests

This section explains the existing data structures in ADCL and discusses the extension towards the timer object taking an iterative TFQMR solver with Jacobi preconditioning as example. One iteration of the solver requires six neighborhood communications to calculate the necessary matrix-vector products and two allreduce operations for scalar products. ADCL should be used in this scenario to tune both the neighborhood communication as well as the allreduce operations.

From the user's point of view, each communication operation is an `ADCL_Request`, a combination of data, a description of which parts of the data to communicate, a process topology and a communication pattern. Thus, the neighborhood communications are replaced by requests 1–6 and the two allreduce operations form requests 7 and 8 as depicted in fig. 6.3. Internally, the `ADCL_Request` is represented by two objects: *(i)* a request object, which contains information how to communicate the data build form the data, data description and topology and *(ii)* an emethod object which possesses information about the search and decision process. The division allows that requests share a common emethod object if they only differ in the actual data but have the same operation, the same communication volume and process topology. These requests can prune the search space together, thus combine their performance data. This is the case for the neighborhood communication as well as for the allreduce operations, so requests 1–6 share one emethod object as well as requests 7–8.

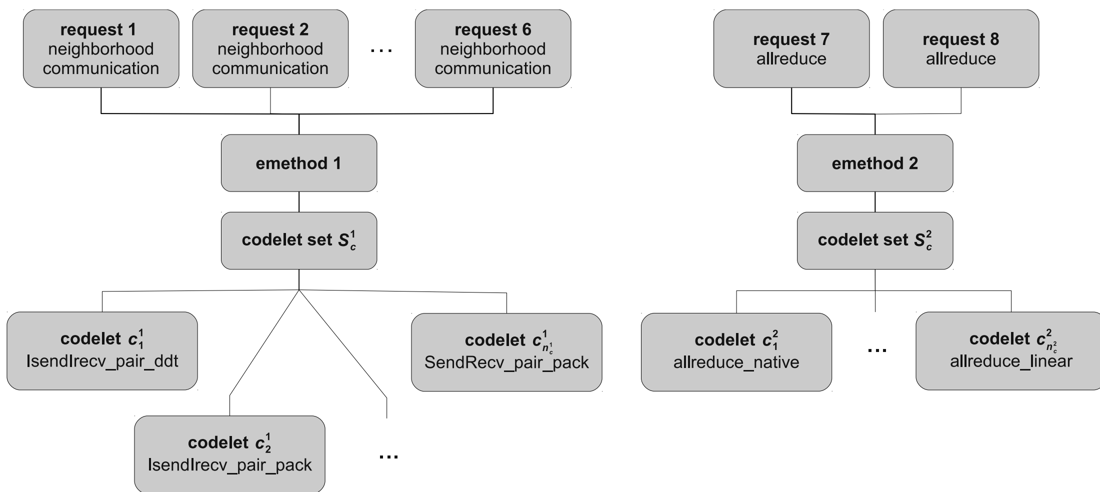


Figure 6.3: Simplified diagram of the dependencies of ADCL objects for the TFQMR solver without timer object.

Each emethod object has a set of codelets which represents the search space. For the neighborhood communication, the predefined set S_c^1 of $n_c^1 = 20$ codelets c_i^1 presented in [103] is taken. For the allreduce operations the predefined set S_c^2 with currently $n_c^2 = 5$ codelets c_i^2 was used (cf. sec. 4.1.3). A set of codelets is thus denoted as $S_c^k = \left\{c_1^k, \ldots, c_{n_c^k}^k\right\}$.

A timer object should allow to optimize neighborhood communication and allreduce operations simultaneously. It might be possible that choosing a certain codelet for the neighborhood communication influences the performance of the allreduce operation and vice versa. Therefore all possible combinations of different codelets from the codelet sets S_c^1 and S_c^2 should be evaluated. To allow for code reuse, one has to keep in mind that major parts of ADCL's empirical optimization process, i.e. pruning the search space, involves handling codelets: the functionality to determine which codelet to evaluate next, to execute the codelet and to evaluate the performance data.

To follow this concept, meta data structures are introduced which combine information from different requests. A meta-codelet is a tuple

$$c_{i+(j-1)n_c^2}^{\text{meta}} = [c_i^1,\, c_j^2],$$

and the meta-set of codelets, which contains all possible combinations of codelets from the codelet sets S_c^1 and S_c^2, is

$$S_c^{\text{meta}} = S_c^1 \times S_c^2 = \left\{[c_i^1, c_j^2], \quad i = 1, \ldots, n_c^1;\, j = 1, \ldots, n_c^2\right\},$$

containing for this example $20 \cdot 5 = 100$ codelets.

As shown in fig. 6.4, the timer object is associated with requests 1–8. It controls the execution of a meta-codelet by assigning the right parts of the tuple to the corresponding requests. The timer object has its own emethod object to collect and process the empirical data. The attached meta-codelet set and the meta-codelets possess the same structure as a traditional codelet set and codelets, allowing to reuse large parts of the functionality of ADCL.

For the sake of completeness it is mentioned that each codelet has attribute sets with attribute values that are used for the attribute-based search. To allow for an attribute-based search with multiple requests, meta-attributes and meta-attribute sets have been constructed in the same way as for the codelets and sets of codelets.

6.2.3 Integration of the Timer Object

The ADCL implementation of the FFT (see sec. 4.2.1 fig. 4.1) is extended to integrate the timer object. The modified listing is shown in fig. 6.5. The timer

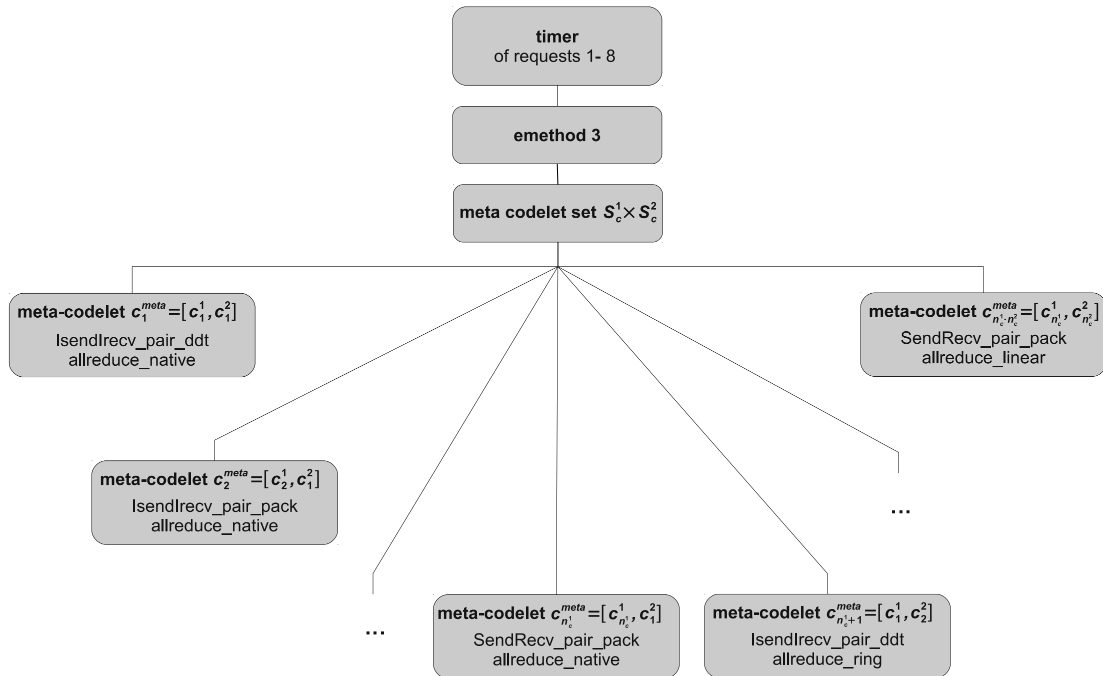


Figure 6.4: Simplified diagram of the dependencies of ADCL objects for the TFQMR solver with timer object.

object is created with `ADCL_Timer_create` which takes a list of ADCL requests as argument. During the creation of the timer, this list of requests (and thus emethods) is associated with the timer object, meta-data structures are set up and dedicated storage space for the empirical data collected by the timer object is allocated. When the timer is started, it inquires a meta-codelet and assigns the codelets of the tuple to the corresponding requests which will later execute them. The associated requests are no longer allowed to report their own performance data. As before, the data exchange is initiated by the `ADCL_Request_start` function. When the timer is stopped, it stores the execution time of the code section embraced by `ADCL_Timer_start` and `ADCL_Timer_stop`. During the deletion of the timer, the timer is uncoupled from the requests and its data structures are freed.

An additional integer variable `adcl_timer` has been added in `global.h`. The code of the subroutine `transpose2_global` stays the same.

```
 program ft
 include 'ADCL.inc'
c  FT: further includes and declarations
  call MPI_Init(ierr)
  call ADCL_Init(ierr)
c FT: timing and setup

c set up ADCL data structures for send and receive vectors, vmap,
c topology and request

c define timer object
  call ADCL_Timer_create ( 1, adcl_request, adcl_timer, ierr )

c main loop
  do iter = 1, niter
      call ADCL_Timer_start ( adcl_timer, ierr )

      call evolve(...)
      call fft(...) ! calls transpose_xy_z which calls transpose2_global

      call ADCL_Timer_stop ( adcl_timer, ierr )
  end do

c FT: verification and output

c free timer object
  call ADCL_Timer_free ( adcl_timer, ierr )
c free other ADCL objects

  call ADCL_Finalize(ierr)
  call MPI_Finalize(ierr)
  end program FT
```

Figure 6.5: Modifications of the ADCL implementation of the FFT NAS Parallel Benchmark when using the timer object.

6.3 Performance Evaluation

The different timing techniques for collective communications are evaluated with the FT benchmark introduced in sec. 2.4. The test systems used are the *Laki* cluster, the Cray XT5m and the NEC-SX8 installation at HLRS, the SGI Altix at LRZ Munich and the Blue Gene/P system at the Supercomputing System Jülich. Runs were executed in the virtual node mode, i.e. every core ran an MPI process.

Within a single batch job, $n_r = 3$ sets of runs are executed. Each set contains 12 runs, 4 runs for the different timing techniques and one verification run for each of the 8 codelets c_0–c_7 for the all-to-all operation (cf. sec. 4.1.3). For the former, 200 FFT iterations are used. To evaluate the timing techniques, each of the eight codelets is measured 20 times which corresponds to 20 FFT iterations. For *timer_multistep*, n_s was set to 4 entailing $m = 5$.

6.3.1 Ranking the codelets from best to worst

Our first concern is the informational value of the verification runs when trying to establish a ranking of the codelets from best to worst. Collectives are known to have the potential to cause network flooding. This fact by itself can produce large deviations and it worsens if resources are shared. Limited influence on the process placement or a poor implementation of parts of the MPI library can aggravate the network effects. This topic was already discussed in a slightly different setting in sec. 5.3.3.1.

Figure 6.6 shows an example where codelet c_7 could be ranked anywhere from first (if the execution times of c_0 and c_6 are large enough) to 8th (if the execution time of c_1 is small enough). One might argue that a way out would be to increase the number of FFT iterations, thus to prolong the duration of the verification runs until a unique ranking can be established. This rises the question which time span leads to stability and thus introduces an unknown parameter as the amount of increase necessary is not known beforehand. It can only be determined by trying out different values which leads us back to traditional benchmarking and contradicts the principle of automatic tuning. As a consequence also the number of measurements per codelet during the testing phase would have to be increased and prolong running maybe less efficient codelets. If a ranking can not be established for long term runs, it would be foolish to require this during the search phase. To sum up, large deviations between the verification runs are possible and they lie outside the control of the HPC user.

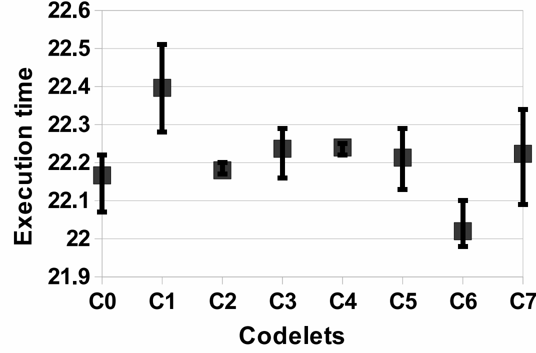


Figure 6.6: Example for instable verification runs. Average execution time. Y error bars show range of execution time of the 3 verification runs.

So a new measure of instability needs to be introduced, since the sample standard deviation is inapplicable to quantify the deviations in this context. Suppose that all codelets have the nearly exact same execution time for each of the three verification runs resulting in a comparable standard deviation which is assumed to be very small. Although the latter is small, there is no possibility to establish any ranking. Let $t_v^i(c), i = 1, \ldots, n_r$ denote the verification run times of a codelet c and $t_v^{\text{avg}}(c)$, $t_v^{\text{min}}(c)$, and $t_v^{\text{max}}(c)$ the average, minimum and maximum verification run time of the codelet as defined in eqns. (5.8)–(5.10). Again, operations over i like $\min_i$ abbreviate $\min_{i=1,n_r}$ and over a codelet c like $\sum_c$ stands for $\sum_{c \in \{c_0, \ldots, c_7\}}$.

The instability I

$$I(c) = \frac{1}{n_c - 1} \sum_{d \neq c} o(c, d)$$

characterizes how many codelets could be swapped in a ranking, where the overlap $o(c, d)$ for a codelet c with a codelet d is defined as in eqn. (5.11).

The instability is categorized into 5 classes:

$0 \leq I < 0.2$	very stable	++
$0.2 \leq I < 0.4$	stable	+
$0.4 \leq I < 0.6$	fair	o
$0.6 \leq I < 0.8$	unstable	-
$0.8 \leq I \leq 1$	very unstable	- -

Table 6.1 shows the stability properties of the different test cases.

system MPI	n_p K	Instability I of codelet								$\frac{\sum I}{n_c}$	
		c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7		
laki	8A	0.57	0.86	0.43	0.14	0.29	0.57	0.43	0.14	0.43	o
impi	8B	0.43	0.43	0.43	0.43	0.43	0.43	0.43	0.43	0.43	o
	32B	1.00	0.57	1.00	1.00	0.57	0.57	0.57	0.43	0.71	-
	32C	0.29	0.14	0.57	0.29	0.14	0.43	0.43	0.29	0.32	+
	128C	0.29	0.14	0.29	0.29	0.14	0.00	0.00	0.00	0.14	++
	512C	0.00	0.14	0.00	0.00	0.14	0.00	0.00	0.00	0.04	++
laki	8A	1.00	1.00	1.00	0.86	0.71	0.86	1.00	1.00	0.93	- -
ompi	8B	1.00	1.00	0.86	0.86	1.00	0.71	1.00	1.00	0.93	- -
	32B	0.43	0.29	0.43	0.29	0.29	0.43	0.14	0.29	0.32	+
	32C	0.29	0.14	0.29	0.29	0.14	0.00	0.14	0.14	0.18	++
	128C	0.29	0.14	0.14	0.14	0.14	0.00	0.00	0.00	0.11	++
	512C	0.29	0.14	0.43	0.00	0.14	0.43	0.43	0.14	0.25	+
sgi	8A	0.00	0.43	0.57	0.29	0.29	0.14	0.00	0.29	0.25	+
altix-	8B	0.00	0.86	0.86	0.71	0.71	0.86	0.86	0.86	0.71	-
mpi	32B	0.29	0.14	0.71	0.29	0.14	0.71	0.43	0.43	0.39	+
	32C	0.00	0.14	0.00	0.29	0.14	0.14	0.43	0.29	0.18	++
	128C	0.00	0.14	0.43	0.57	0.14	0.43	0.43	0.14	0.29	+
	256C	0.29	0.29	0.00	0.29	0.43	0.14	0.29	0.57	0.29	+
sgi	8A	0.14	0.29	0.00	0.14	0.29	0.00	0.00	0.29	0.14	++
impi	8B	1.00	1.00	1.00	1.00	1.00	0.86	1.00	0.86	0.96	- -
	32B	0.14	0.14	0.14	0.14	0.14	0.14	0.14	0.14	0.14	++
	32C	0.14	0.14	0.43	0.14	0.14	0.43	0.43	0.43	0.29	+
	128C	0.43	0.14	0.43	0.43	0.14	0.57	0.14	0.00	0.29	+
	256C	1.00	0.29	0.43	0.57	0.29	0.57	0.43	0.71	0.54	o
sgi	8A	1.00	1.00	1.00	0.86	0.86	1.00	1.00	1.00	0.96	- -
ompi	8B	0.71	1.00	0.86	1.00	0.71	1.00	0.86	1.00	0.89	- -
	32B	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	- -
	32C	0.29	0.57	0.71	1.00	0.57	0.29	0.43	0.71	0.57	o
	128C	0.57	0.14	0.43	0.57	0.14	0.29	0.43	0.00	0.32	+
cray	8A	0.29	0.00	0.00	0.14	0.29	0.14	0.29	0.00	0.14	++
native	8B	0.43	0.57	0.14	0.14	0.43	0.43	0.71	0.29	0.39	+
	32B	0.14	0.14	0.29	0.43	0.14	0.29	0.00	0.00	0.18	++
	32C	0.14	0.14	0.00	0.14	0.14	0.00	0.00	0.00	0.07	++
	128C	0.14	0.14	0.14	0.14	0.14	0.14	0.00	0.00	0.11	++
	256C	1.00	0.71	0.57	0.43	0.71	1.00	0.86	0.71	0.75	-
sx8	8A	0.14	0.57	0.71	0.57	0.86	0.57	1.00	0.43	0.61	-
native	8B	0.43	1.00	0.57	0.57	0.57	0.86	0.57	0.29	0.61	-
	16B	0.57	0.14	0.43	0.29	0.14	0.43	0.29	0.00	0.29	+
	32B	0.71	0.14	0.29	0.14	0.14	0.29	0.57	0.29	0.32	+
	32C	0.00	0.00	0.43	0.00	0.00	0.29	0.43	0.29	0.18	++
jugene	8A	0.00	0.29	0.00	0.00	0.29	0.00	0.00	0.29	0.11	++

to be continued on next page

system MPI	n_p K	Instability I of codelet								$\frac{\sum I}{n_c}$	
		c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7		
native	32B	0.00	0.14	0.00	0.00	0.14	0.00	0.00	0.00	0.04	++
	32C	0.00	0.00	0.00	0.00	0.00	0.14	0.29	0.14	0.07	++
	128C	0.00	0.14	0.00	0.00	0.14	0.00	0.00	0.00	0.04	++
	512C	0.00	0.14	0.00	0.00	0.14	0.00	0.00	0.00	0.04	++

Table 6.1: Stability of the verification runs (n_p - number of processes, K - NAS Parallel Benchmark class (see tbl. 2.4), n_c - number of codelets)

Remarkably are the very stable conditions on Blue Gene which are probably a result of the scheduling policy which requires the reservation of a whole blade even for runs with smaller process numbers. Smaller test cases, especially 8B, tend to be less stable than larger test cases. Also the choice of the MPI implementation influences the stability. On *Laki*, the intra-node communication for test cases 8A and 8B with OpenMPI is less stable than with Intel MPI. On the SGI, the native MPI implementation, Intel MPI and OpenMPI each show different stability properties for one test case. The first two MPI implementations produce nearly equally stable measurements whereas OpenMPI tends to be unstable.

6.3.2 Assessment of the timing techniques

With the notions of overlap and instability, we can now assess the accuracy of the timing technique *barrier* (B), *nobarrier* (NB), *timer* (T) and *timer_multistep* (TM) by comparing the rankings of codelets of the timing techniques with the ranking established by the verification runs. For completely stable runs, the rankings of the timing techniques and the verification runs should be the same. Otherwise, at least two codelets overlap. In this case, a flip between the codelets in the ranking is acceptable. However, if two codelets in the ranking do *not* overlap, but are flipped in the ranking of a timing technique, a codelet is rated better than it actually is. To quantify the severity of such a false ordering weight tables are defined. A high instability signifies on the one hand, that the establishment of a ranking is eventually more challenging, on the other hand, it reduces the possibilities for a false ordering. For a completely instable test case, for example, no false ordering is possible since any ranking would be correct.

Based on the empirical data generated during one verification run by a timing technique $M \in \{B, NB, T, TM\}$, an estimated mean $\hat{\mu}^i_M(c)$ is calculated which defines a ranking for each timing technique and every run i.

As the scales of measurements are very different — tens of seconds for the verification runs, seconds for *timer* and *timer_multistep* and microseconds for the *barrier* and *nobarrier* — the overhead is expressed in percentages P instead of absolute times, i.e. all execution times are scaled:

$$\begin{aligned}
P_v^{\mathrm{avg}}(c) &= \frac{t_v^{\mathrm{avg}}(c) - \min_d(t_v^{\mathrm{avg}}(d))}{\min_d(t_v^{\mathrm{avg}}(d))} \cdot 100\%, \\
P_v^{\min}(c) &= \frac{t_v^{\min}(c) - \min_d(t_v^{\min}(d))}{\min_d(t_v^{\min}(d))} \cdot 100\%, \\
P_v^{\max}(c) &= \frac{t_v^{\max}(c) - \min_d(t_v^{\max}(d))}{\min_d(t_v^{\max}(d))} \cdot 100\% \text{ and} \\
P_M^i(c) &= \frac{\hat{\mu}_M^i(c) - \min_d(\hat{\mu}_M^i(d))}{\min_d(\hat{\mu}_M^i(d))} \cdot 100\%, \quad i = 1, \ldots, n_r.
\end{aligned}$$

The average loss or gain between two codelets based on the verification runs defines a weight

$$w_{\mathrm{avg}}(c,d) = (P_v^{\mathrm{avg}}(c) - P_v^{\mathrm{avg}}(d)) \cdot (1 - o(c,d)),$$

and is therefore 0 if a ranking between c and d could not be established or the percental difference of the averaged execution times of the verification runs. The weights derived from the minimum and maximum loss or gain between two codelets based on the verification runs are

$$w_{\min}(c,d) = \begin{cases} (P_v^{\min}(d) - P_v^{\max}(c)) \cdot (1 - o(c,d)) & \text{if } P_v^{\min}(d) - P_v^{\max}(c) < \\ & P_v^{\max}(d) - P_v^{\min}(c) \\ (P_v^{\max}(d) - P_v^{\min}(c)) \cdot (1 - o(c,d)) & \text{otherwise} \end{cases}$$

and

$$w_{\max}(c,d) = \begin{cases} (P_v^{\min}(d) - P_v^{\max}(c)) \cdot (1 - o(c,d)) & \text{if } P_v^{\min}(d) - P_v^{\max}(c) > \\ & P_v^{\max}(d) - P_v^{\min}(c) \\ (P_v^{\max}(d) - P_v^{\min}(c)) \cdot (1 - o(c,d)) & \text{otherwise} \end{cases},$$

respectively.

A timing technique switches two codelets c and d if

$$\operatorname{sgn}(P_M^i(c) - P_M^i(d)) \neq \operatorname{sgn}(P_v^{\mathrm{avg}}(c) - P_v^{\mathrm{avg}}(d)), \tag{6.1}$$

where sgn is the signum function. The gravity of this error is based on the weights w. The average error is

$$e_{\mathrm{avg}}(c,d) = \begin{cases} \max(0, w_{\mathrm{avg}}(c,d)) & \text{if (6.1) holds} \\ 0 & \text{otherwise,} \end{cases}$$

its minimum

$$e_{\min}(c,d) = \begin{cases} \max(0, w_{\min}(c,d)) & \text{if (6.1) holds} \\ 0 & \text{otherwise} \end{cases}$$

and its maximum

$$e_{\max}(c,d) = \begin{cases} \max(0, w_{\max}(c,d)) & \text{if (6.1) holds} \\ 0 & \text{otherwise.} \end{cases}$$

The comparison with 0 avoids counting ranking errors twice which would happen if absolute values are used. This results in three tables of weights for the minimum, average and maximum errors.

Figure 6.7 shows the maximum average loss $\max_{c,d} e_{\text{avg}}(c,d)$ for one test case whereas fig. 6.8 displays the sum of all average losses $\sum_{c,d} e_{\text{avg}}(c,d)$, both averaged over all sets. The range is depicted by error bars. The labels are a combination of HPC system, the MPI used (ompi – OpenMPI, impi – Intel MPI), the number of processes and the NAS parallel benchmark problem class, e.g. `laki_ompi_8A` denotes a class A FT benchmark run with 8 processes using OpenMPI on the *Laki* Nehalem cluster.

In most cases, the largest errors in the ranking are in the range of 0 up to 5% overhead. There is no clear dependence of errors or error ranges on the instability, but all timing techniques tend to have more problems the larger I gets. The results show that *barrier* and *nobarrier* have trouble finding the adequate ranking. *timer* shows nearly the exact same behavior, whereas *timer_multistep* proves to be very effective on the SGI Altix. Only for `cray_256C` *timer_multistep* does not at all perform well. The latter might be attributed to the fact that only batchjob has been executed for each tuple (HPC system, MPI implementation, number of processes, benchmark class). The reasons are that results from multiple batchjobs are not easily comparable due to different process placements and a sufficient number of batchjobs to obtain a valid statistic would have exceeded our computing time budgets. Hence there is the possibility that runtime conditions for a tuple were stable, but did not show the normal behavior. This is an exception, so even if one tuple shows a misleading behavior of the timing techniques, taking into account all tuples still gives a good picture of the performance of the techniques.

The maximum ranking error has only some informational value regarding the performance of ADCL: firstly, the percentage error $(P^i_M(c) - P^i_M(c_{\text{winner}}))(1 - o(c, c_{\text{winner}}))$ between one codelet and the winner codelet is in practice always smaller than the maximum average ranking error $e_{\text{avg}}(c, c_{\text{winner}},)$ depicted in fig. 6.7 since a misclassification does not necessarily involve the winner codelet. Secondly, even if the ranking error involves the winner codelet, i.e. the best-performing codelet is not recognized, it is still possible that ADCL is faster than the native MPI implementation.

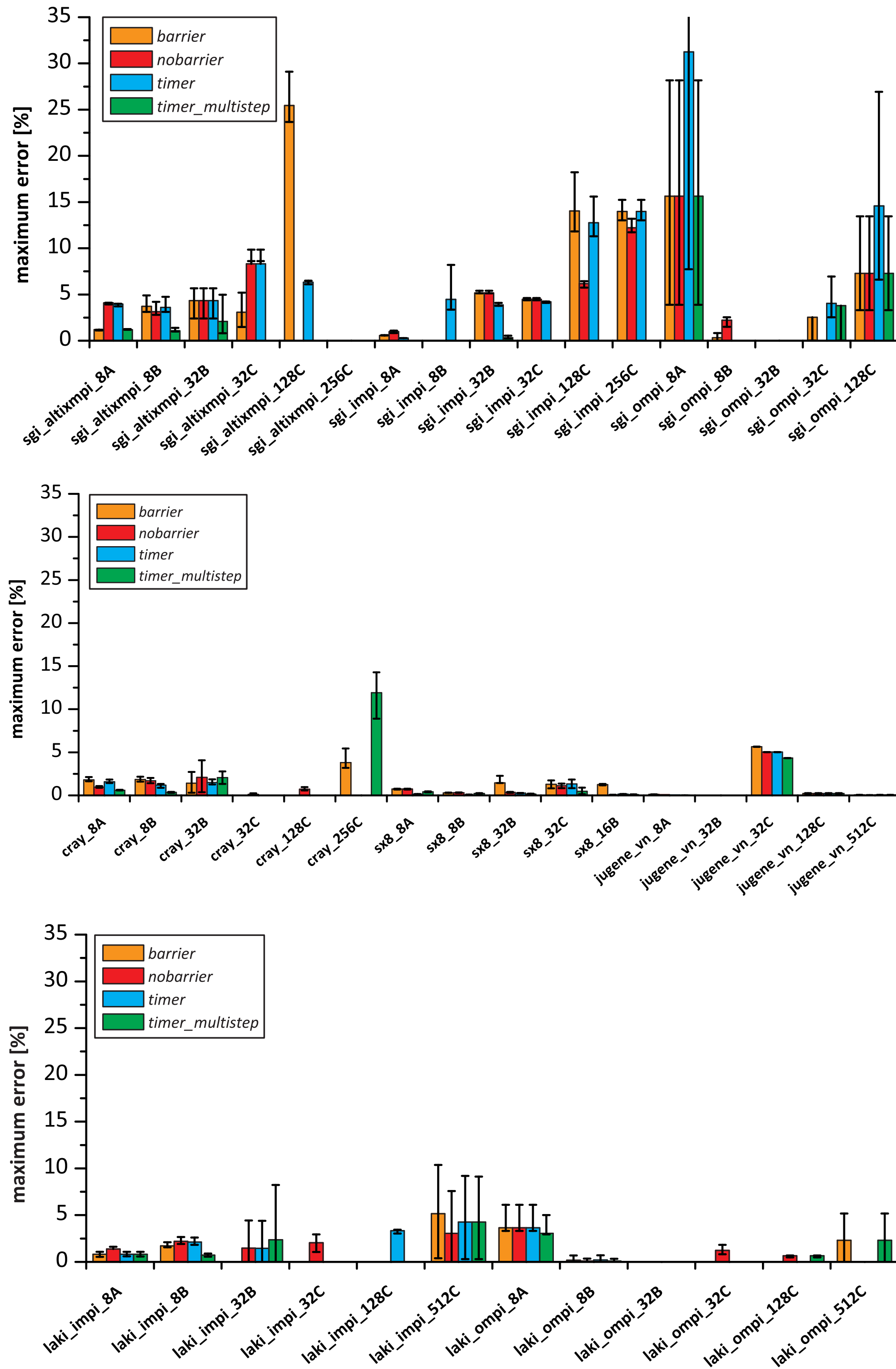


Figure 6.7: Maximum average error $\max_{c,d} e_{\text{avg}}(c,d)$ between two codelets c and d for different test cases. Y error bars show range from $\max_{c,d} e_{\text{min}}(c,d)$ to $\max_{c,d} e_{\text{max}}(c,d)$ between the three verification runs.

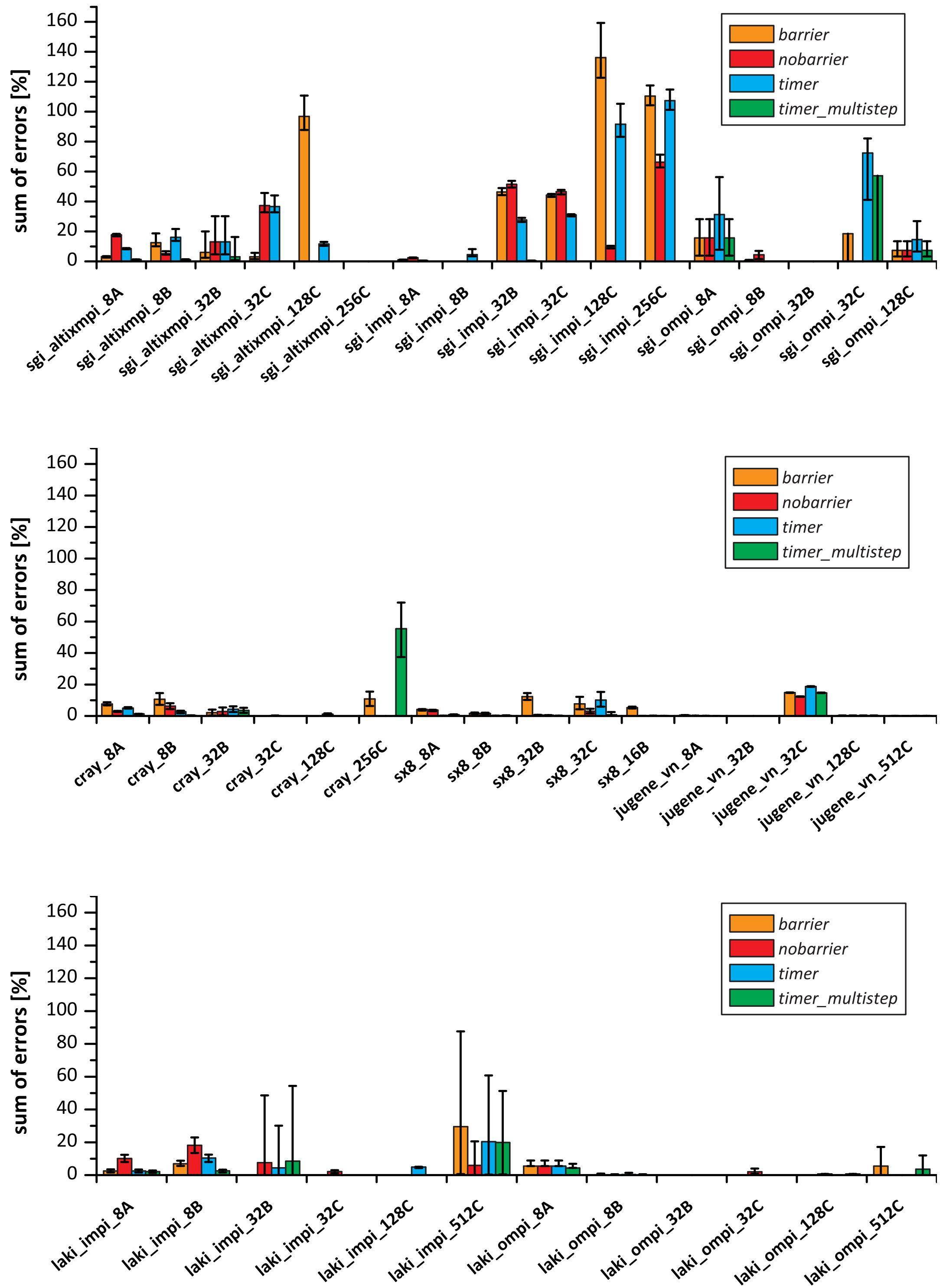


Figure 6.8: Sum of all average errors $\sum_{c,d} e_{\text{avg}}(c,d)$ between two codelets c and d for different test cases. Y error bars show range from $\sum_{c,d} e_{\text{min}}(c,d)$ to $\sum_{c,d} e_{\text{max}}(c,d)$ between the three verification runs.

6.4 Conclusion

This chapter introduced the timer object within the Abstract Data and Communication Library (ADCL) which allows to time not only a single codelet but also its environment. It separates timing of the communication and its actual execution. This allows to tune two complex scenarios that could not be properly handled up to now, namely tuning multiple communication operations simultaneously in case of (suspected) dependencies between these operations, and tuning code segments that overlap communication and computation. More generally, through the conceptual separation any code portions of interest can be instrumented for measurements. The timer object represents however more than just the timing itself, since it allows (a) to combine the set of codelets of all operations to be tuned, and all according internal functions like an attribute-based search and (b) it takes control of the tuning and coordinates the execution of all requests registered at creation.

Comparing four different timing techniques to produce data for the empirical optimization process showed the benefits of this approach. Using the timer object in multi-step mode—and thus execution times of larger code portions as a basis for the empirical optimization—works especially well on the SGI Altix and produces good results on NEC SX-8, *Jugene*, the *Laki* cluster and in general on the Cray XT-5m. It clearly improves the accuracy of the results compared to a highly synchronized timing (i.e. without the timer object) or with no synchronization at all.

7 Conclusions and Outlook

Empirical optimization libraries open up new vistas on the problem of laborious and error-prone hand-tuning of simulation code. They enable an end-user to use high-performance computing resources efficiently without special knowledge and extensive time effort. This is especially important in the field of computational fluid dynamics (CFD) where high resolution simulations require large-scale parallel simulations. Without such libraries, the variety of test cases, runtime parameters as well as the supercomputers' hard- and software configuration would demand an on-going effort in benchmarking and tuning.

The Abstract Data and Communication Library (ADCL) is an application level optimization library with special focus on tuning communication operations in MPI parallel simulations. It provides its own API which separates the user's communication request from the underlying communication layer. This has several advantages for the end-user as well as from a design perspective. Predefined function sets, e.g. for Cartesian neighborhood communication operations, simplify parallelization and optimization tasks for the end-user. Only the ADCL objects for data and topology have to be initialized. The MPI communication itself is hidden inside ADCL. This reduces the programming effort and facilitates an initial MPI implementation. The abstract interfaces of ADCL allow not only to tune MPI communications but to replace the underlying communication layer by other mechanisms. GASNET or PGAS languages are two possibilities.

This work presented enhancements to the Abstract Data and Communication Library (ADCL) to make it applicable to CFD simulations:

The first result of this thesis widens the area of application of ADCL. The separation of data and data description using the introduced ADCL vector map (vmap) object extended the ADCL from neighborhood communication patters to collective operations. Predefined sets of codelets for `allreduce`, `allgathv` and `alltoall` have been implemented to assist the user in the optimization process. This extension made the tuning of the FFT kernel within spectral CFD methods possible. The performance of the ADCL-enriched FFT benchmark was tested in a variety of different settings. The integration of ADCL showed major performance improvements in six of the test cases where the native `MPI_Alltoall` implementation delivered a non-optimal performance, and minor

improvements in the majority of test cases. In only three test cases a slight decrease of the performance was observed.

The second result constitutes an important achievement towards improved evaluation of the empirical data. The quality of four different methods to handle outliers were analyzed in order to choose the best-performing codelet: a heuristic derived from the trimmed mean value, a standard approach using interquartile ranges, a technique based on cluster analysis and an approach including robust statistics. Their performance was evaluated using the structured Finite Volume code Euler3D which exhibits a next-neighbor communication pattern. Together with other investigations for a TFQMR solver in [102], the heuristic presented itself as the most reliable out of the four methods and delivered optimal or close to optimal performance in most of the test cases. It is also favorable due to its low computational complexity. The method based on robust statistics was the most successful determining subtle differences between implementation alternatives, but failed—as did the method based on cluster analysis—for the TFQMR example for data which is more widespread. Also, the method using interquartile ranges showed an mediocre accuracy in this case. It was shown that ADCL could improve the applications performance up to 33%, if the right codelets are selected.

As the third result, the reliability of empirical data collection could be greatly improved with the design and implementation of an ADCL timer object. A specific timing technique based on this object, the *timer_multistep*, showed a substantial decrease in misclassifications when ranking the codelets. The newly introduced ADCL timer object makes it now possible to tune multiple MPI communications independently as well as code with overlapping communication and computation.

ADCL has been applied in large-scale CFD computations and showed that it can tremendously improve the applications performance. The user benefits from the abstract interface which allows him to achieve portable performance using an easy description of the communication operation. He/She does not have to bother with MPI performance issues whenever he/she first uses or changes anything in the hard- or software environment such as HPC system, test case or MPI implementation. ADCL is very beneficial for benchmarking or optimizations. It can also provide an easy means for MPI vendors or system administrators to evaluate the quality of MPI implementations.

This research can be continued in various directions: current efforts include using historic learning to shorten the selection process, to integrate new search methods such as 2^k factorial design or an early stopping criterion and to extend ADCL to other CFD applications. Predefined function sets for extend neighborhood communication pattern have to be included which are e.g. needed in

Lattice-Boltzmann simulations. Also, the automatic optimization of neighborhood communications on unstructured grids needs to be added.

The timer object also offers several interesting research topics: the simultaneous optimization of multiple, dependent communications as outlined e.g. in the TFQMR solver example, the elaboration of the optimization possibilities for code with overlapping computations and communications as well as attribute-based selection algorithms with combined attributes from multiple requests. The latter can also include research on methods to automatically determine the impact of certain attributes on the performance and its handling.

Bibliography

[1] Ruprecht, A., Eisinger, R., Göde, E., Rainer, D.: Virtual Numerical Test Bed for Intuitive Design of Hydro Turbine Components. In: Hydropower into the Next Century, Gmunden, Austria (1999)

[2] Giesecke, J., Mosonyi, E., Heimerl, S.: Wasserkraftanlagen: Planung, Bau und Betrieb. 5 edn. Springer, Berlin (2009)

[3] Ye, Q., Domnick, J., Scheibe, A., Pulli, K.: Numerical Simulation of Electrostatic Spray-painting Processes in the Automotive Industry. In Krause, E., Jäger, W., Resch, M., eds.: High Performance Computing in Science and Engineering 2004 Transactions of the High Performance Computing Center Stuttgart (HLRS) 2004, Springer Berlin Heidelberg (2004) 261–275

[4] Andreassi, L., Facci, A.L., Ubertini, S.: Three-Dimensional Simulation of Gaseous Fuel Injection Through a Hybrid Approach. Journal of Engineering for Gas Turbines and Power **132**(7) (2010) 074502

[5] Cavanagh, E.J.: Simulation of automotive fuel injection equipment. Doctoral thesis, University of Southampton (2008)

[6] Baker, P.J., Jenkins, M.D.: An Optimised Thermal Design and Development Process for Passenger Compartments of Vehicles (2009)

[7] Cullimore, B.A., Hendricks, T.J.: Design and Transient Simulation of Vehicle Air Conditioning Systems (2001)

[8] Reneaux, J.: Overview on Drag Reduction Technologies for Civil Transport Aircraft. In Neittaanmäki, P., Rossi, T., Korotov, S., Oñate, E., Périaux, J., Knörzer, D., eds.: Proceedings of the European Congress on Computational Methods in Applied Sciences and Engineering (ECCOMAS), 24-28 July 2004, Jyväskylä. Number July (2004) 1–18

[9] Furumura, T.: Parallel Simulation of Seismic Wave Propagation. In Goos, G., Harmanis, J., van Leeuwen, J., eds.: High Performance Computing, Proceedings of the 4th International Symposium, ISHPC 2002 Kansai Science City, Japan, May 15-17, 2002, Springer Berlin / Heidelberg (2002) 373–378

[10] Fertig, M., Petkow, D., Stindl, T., Quandt, M., Munz, C.D., Neudorfer, J., Roller, S., D'Andrea, D., Schneider, R.: Hybrid Code Development for the Numerical Simulation of Instationary Magnetoplasmadynamic Thrusters. In: High Performance Computing In and Engineering \'08, Springer Berlin, Heidelberg (2009) 585–597

[11] Biastoch, A.: The Agulhas Leakage : Role of Mesoscale Processes and Impact on the Atlantic Meridional Overturning Circulation. Habilitationsschrift, Christian-Albrechts-Universität zu Kiel (2008)

[12] Currle-Linde, N., Küster, U., Resch, M., Risio, B.: Science Experimental Grid Laboratory (SEGL) - Dynamical Parameter Study in Distributed Systems. In Joubert, G.R., Nagel, W.E., Peters, F.J., Plata, O., Tirado, P., Zapata, E., eds.: Parallel Computing: Current & Future Issues of High-End Computing, Proceedings of the International Conference ParCo 2005. Volume 33., NIC Series, Vol. 33, John von Neumann Institute for Computing J\"ulich (2006) 49–56

[13] Orszag, S.A., Patterson, G.S.: Numerical Simulation of Three-Dimensional Homogeneous Isotropic Turbulence. Physical Review Letters **28**(2) (January 1972) 76–79

[14] Yokokawa, M., Itakura, K., Uno, A., Ishihara, T., Kaneda, Y.: 16.4-Tflops Direct Numerical Simulation of Turbulence by a Fourier Spectral. In: Proceedings of the IEEE/ACM SC2002 Conference (CD-ROM), Baltimore. Volume 00.

[15] Wulf, W., McKee, S.: Hitting the memory wall: Implications of the obvious. ACM SIGARCH Computer Architecture News **23**(1) (March 1995) 20–24

[16] McCalpin, J.: Memory bandwidth and machine balance in current high performance computers (December 1995)

[17] Callahan, D., Carr, S., Kennedy, K.: Improving Register Allocation for Subscripted Variables. SIGPLAN Not. **25**(6) (1990) 53–65

[18] Message Passing Interface Forum: MPI: A Message-Passing Interface Standard, Version 2.2. High Performance Computing Center Stuttgart (HLRS) (2009)

[19] Lawson, C.L., Hanson, R.J., Kincaid, D.R., Krogh, F.T.: Basic linear algebra subprograms for Fortran usage. ACM Transactions on Mathematical Software (TOMS) **5**(3) (1979) 308–323

[20] Dongarra, J., Du Croz, J., Hammarling, S., Hanson, R.: An extended set of FORTRAN basic linear algebra subprograms. ACM Transactions on Mathematical Software (TOMS) **14**(1) (March 1988) 1–17

[21] Dongarra, J., Du Croz, J., Hammarling, S., Duff, I.: A set of level 3 basic linear algebra subprograms. ACM Transactions on Mathematical Software (TOMS) **16**(1) (March 1990) 1–17

[22] Blackford, L.S., Sorensen, D., Anderson, E.: LAPACK User's Guide. Society for Industrial & Applied Mathematics (2000)

[23] Swarztrauber, P.N.: Vectorizing The FFTs. In Rodrigue, G., ed.: Parallel Computations, New York, Academic Press (1982) 51–83

[24] Rabenseifner, R.: Automatic mpi counter profiling. In: 42nd CUG Conference, Noorwijk, The Netherlands., Citeseer (2000)

[25] Resch, M.M.: Remarks on Supercomputing in Germany. PIK Praxis der Informationsverarbeitung und Kommunikation **28** (2005) 238–242

[26] Clint Whaley, R., Petitet, A., Dongarra, J.: Automated empirical optimizations of software and the ATLAS project. Parallel Computing **27**(1-2) (January 2001) 3–35

[27] Roe, P.L.: Approximate Riemann solvers, parameter vectors and difference schemes. Journal of Computational Physics **43**(2) (1981) 357–372

[28] Orszag, S.A.: Numerical simulation of turbulence. Physics of Fluids Suppl. II **12** (1969) 250–257

[29] Boyd, J.P.: Chebyshev and Fourier Spectral Methods. 2 edn. Number 2. DOVER Publications, Inc. (2000)

[30] Canuto, C., Hussaini, M., Quarteroni, A., Zang, T.: Spectral Methods: Fundamentals in Single Domains. 3 edn. Scientific Computation. Springer (2006)

[31] Canuto, C., Hussaini, M., Quarteroni, A., Zang, T.: Spectral Methods: Evolution to Complex Geometries and Applications to Fluid Dynamics. Scientific Computation. Springer (2007)

[32] Cooley, J.W., Tukey, J.W.: An Algorithm for the Machine Calculation of Complex Fourier Series. Mathematics of Computation **19**(90) (1965) 297–301

[33] Grama, A., Karypis, G., Kumar, V., Gupta, A.: Introduction to Parallel Computing. 2 edn. Addison Wesley (2003)

[34] Fox, G.C., Johnson, M.A., Lyzenga, G.A., Otto, S.W., Salmon, J.K., Walker, D.W.: Solving problems on concurrent processors. Vol. 1: General techniques and regular problems. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1988)

[35] Wang, X.: Kopplungsframework für Multi-Scale Gitter (2008)

[36] Bailey, D., Barszcz, E., Barton, J., Browning, D., Carter, R., Dagum, L., Fatoohi, R., Fineberg, S., Frederickson, P., Lasinski, T., Schreiber, R., Simon, H., Venkatakrishnan, V., Weeratunga, S.: {The NAS Parallel Benchmarks} (1994)

[37] Bailey, D., Harris, T., Saphir, W., van Der Wijngaart, R., Woo, A., Yarrow, M.: The NAS Parallel Benchmarks 2.0 (1995)

[38] Saphir, W., van Der Wijngaart, R., Woo, A., Yarrow, M.: New Implementations and Results for the NAS Parallel Benchmarks 2. In: Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing, PPSC 1997, March 14-17, 1997, Minneapolis, Minnesota, USA, NASA Advanced Supercomputing (NAS), SIAM (1997)

[39] Blackford, L., Petitet, A., Pozo, R., Remington, K., Whaley, R., Demmel, J., Dongarra, J., Duff, I., Hammarling, S., Henry, G.: An updated set of basic linear algebra subprograms (BLAS). ACM Transactions on Mathematical Software **28**(2) (June 2002) 135–151

[40] Bilmes, J., Chin, C.W., Demmel, J., Lam, D., Asanović, K.: Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology (1996)

[41] Frigo, M., Johnson, S.: The fastest fourier transform in the west (1997)

[42] See homepage for details: ATLAS home page (2010) http://math-atlas.sourceforge.net/.

[43] Whaley, R., Dongarra, J.: Automatically tuned linear algebra software (December 1997)

[44] Whaley, R., Dongarra, J.: Automatically tuned linear algebra software. In: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM), San Jose, CA, IEEE Computer Society Washington, DC, USA (1998) 1–27

[45] Püschel, M., Moura, J., Singer, B., Xiong, J., Johnson, J., Padua, D., Veloso, M., Johnson, R.: Spiral: A generator for platform-adapted libraries of signal processing algorithms. Journal of High Performance Computing Applications **18**(1) (2004) 21–45

[46] Püschel, M., Moura, J., Johnson, J., Padua, D., Veloso, M., Singer, B., Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., Others: SPIRAL: Code generation for DSP transforms. Proceedings of the IEEE **93**(2) (2005) 232–275

[47] Püschel, M., Singer, B., Veloso, M., Moura, J.: Fast automatic generation of DSP algorithms. In: Lecture Notes in Computer Science. Volume 2073 of LNCS., San Francisco, CA, USA, Springer (May 2001) 97–106

[48] Singer, B., Veloso, M.: Learning to Predict Performance from Formula Modeling and Training Data. In: In Proceedings of the Seventeenth International Conference on Machine Learning, Morgan (2000) 887–894

[49] Mirkovic, D., Johnsson, S.: Automatic performance tuning in the UHFFT library. In: Lecture notes in computer science, Springer (2001) 71–80

[50] Dongarra, J., Eijkhout, V.: Self-adapting numerical software for next generation applications (2002)

[51] Chen, Z., Dongarra, J., Luszczek, P., Roche, K.: Self-adapting software for numerical linear algebra and LAPACK for clusters. Parallel Computing **29**(11-12) (January 2003) 1723–1743

[52] Dongarra, J., Eijkhout, V.: Self-Adapting Numerical Software and Automatic Tuning of Heuristics (2003)

[53] Chen, Z., Dongarra, J., Luszczek, P., Roche, K.: The LAPACK for clusters project: an example of self adapting numerical software. In: Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS 04'). Volume Track 9., Big Island, Hawaii (January 2004) 90282a

[54] Liniker, P., Beckmann, O., Kelly, P.: Delayed evaluation, self-optimising software components as a programming model. In: Lecture notes in computer science, Paderborn, Germany, Springer (2002) 666–674

[55] Im, E.J.: Optimizing the performance of sparse matrix-vector multiplication. PhD thesis (May 2000)

[56] Im, E., Yelick, K., Vuduc, R.: Sparsity: Optimization framework for sparse matrix kernels. International Journal of High Performance Computing Applications **18**(1) (2004) 135

[57] Duff, I.S., Heroux, M.A., Pozo, R.: An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum. ACM Trans. Math. Softw. **28**(2) (2002) 239–267

[58] Buttari, A., Eijkhout, V., Langou, J., Filippone, S.: Performance optimization and modeling of blocked sparse kernels. International Journal of High Performance Computing Applications **21**(4) (2007) 467–484

[59] Vuduc, R., Demmel, J., Yelick, K.: OSKI: A library of automatically tuned sparse matrix kernels. In: Journal of Physics: Conference Series. Volume 16 of Journal of Physics: Conference Series., San Francisco, CA, USA, Institute of Physics Publishing (June 2005) 521–530

[60] Eijkhout, V., Fuentes, E., Eidson, T., Dongarra, J.: The Component Structure of a Self-Adapting Numerical Software System. International Journal of Parallel Programming **33**(2) (2005)

[61] Vadhiyar, S.S., Dongarra, J.J., Fagg, G.E.: ACCT: Automatic Collective Communications Tuning. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proceedings of the 7th European PVM/MPI Users' Group Meeting Balatonfüred, Hungary, September 10-13, 2000. Volume 1908/2000 of LNCS., Springer Berlin / Heidelberg (2000) 354–361

[62] Vadhiyar, S., Fagg, G., Dongarra, J.: Automatically tuned collective communications. In: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM), Dallas, Texas, United States, IEEE Computer Society Washington, DC, USA (2000) 3

[63] Faraj, A., Yuan, X., Lowenthal, D.: STAR-MPI: self tuned adaptive routines for MPI collective operations. In: Proceedings of the 20th annual international conference on Supercomputing, Cairns, Queensland, Australia, ACM (2006) 208

[64] Hartmann, O., Kühnemann, M., Rauber, T., Rünger, G.: An adaptive extension library for improving collective communication operations. Concurrency and Computation: Practice and Experience **20**(10) (2008) 1173–1194

[65] Faraj, A., Yuan, X.: Automatic generation and tuning of MPI collective communication routines. In: Proceedings of the 19th annual international conference on Supercomputing, ACM (2005) 402

[66] Evans, J.J., Hood, C.S., Groop, W.D.: Exploring the Relationship between Parallel Application Run-Time Variability and Network Performance in Clusters. (2003) 538–547

[67] Petrini, F., Kerbyson, D.J., Pakin, S.: The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In: SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing, Washington, DC, USA, I.E.E.E.Press (2003) 55

[68] Faraj, A., Patarasuk, P., Yuan, X.: A Study of Process Arrival Patterns for MPI Collective Operations. International Journal of Parallel Programming **36**(6) (2007) 543–570

[69] Hockney, R.W.: The Communication Challenge for {MPP}: Intel {Paragon} and {Meiko CS-2}. Parallel Computing **20**(3) (1994) 389–398

[70] Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K.E., Santos, E., Subramonian, R., von Eicken, T.: {LogP}: Towards a realistic model of parallel computation. In: Proceedings of the fourth {ACM SIGPLAN} symposium on Principles and practice of parallel programming, ACM Press (1993) 1–12

[71] Alexandrov, A., Ionescu, M.F., Schauser, K.E., Scheiman, C.: {LogGP}: Incorporating long messages into the {LogP} model. In: Proceedings of the seventh annual {ACM} symposium on Parallel algorithms and architectures, ACM Press (1995) 95–105

[72] Kielmann, T., Bhoedjang, R.A.F., Plaat, A., Hofman, R.F.H., Bal, H.E.: MAGPIE: MPI's collective communication operations for clustered wide area systems. SIGPLAN Not. **34**(8) (1999) 131–140

[73] Vuduc, R., Demmel, J., Bilmes, J.: Statistical models for empirical search-based performance tuning. International Journal of High Performance Computing Applications **18**(1) (2004) 65

[74] Seymour, K., You, H., Dongarra, J.: A comparison of search heuristics for empirical code optimization. In: Proceedings of the 2008 IEEE International Conference on Cluster Computing (3rd International Workshop on Automatic Performance Tuning). (2008) 421–429

[75] Hoefler, T., Lichei, A., Rehm, W.: Low-Overhead LogGP Parameter Assessment for Modern Interconnect Networks. In: Proceedings of the IPDPS, IEEE (March 2007)

[76] Pješivac-Grbović, J., Bosilca, G., Fagg, G., Angskun, T., Dongarra, J.: MPI collective algorithm selection and quadtree encoding. Parallel Computing **33**(9) (2007) 613–623

[77] Kulkarni, M., Pingali, K.: An Experimental Study of Self-Optimizing Dense Linear Algebra Software. PROCEEDINGS-IEEE **96**(5) (2008) 832

[78] Gabriel, E., Feki, S., Benkert, K., Chaarawi, M.: The Abstract Data and Communication Library. Journal of Algorithms and Computational Technology, Special Issue on Computational Science for Medicine, Energy, and Environment Applications **2**(4) (2008) 581–600

[79] Gabriel, E., Huang, S.: Runtime Optimization of Application Level Communication Patterns. (2007) 1–8

[80] Chaarawi, M., Squyres, J., Gabriel, E., Feki, S.: A tool for optimizing runtime parameters of open mpi. In: Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface. Volume 5205/2008 of Lecture Notes in Computer Science., Dublin, Ireland, Springer (2008) 217

[81] Tabe, T., Stout, Q.: The Use of the MPI Communication Library in the NAS Parallel Benchmark. Technical Report CSE-TR-386-99, Department of Computer Science, University of Michigan (November 1999)

[82] Kerbyson, D., Barker, K.: Automatic Identification of Application Communication Patterns via Templates. In: Proc. 18th International Conference

on Parallel and Distributed Computing Systems (PDCS-2005), Las Vegas, NV (September 2005)

[83] Gabriel, E., Saber, F., Benkert, K., Chaarawi, M. In: Towards Performance and Portability through Runtime Adaption for High Performance Computing Applications. (2008)

[84] Feki, S., Gabriel, E.: Incorporating Historic Knowledge into a Communication Library for Self-Optimizing High Performance Computing Applications, Venice, IEEE Computer Society (2008) 265–274

[85] Jones, T.: Survey of MPI Call Usage. In: IBM System Scientific User Group (ScicomP) 10. (2004)

[86] Thakur, R., Gropp, W., Rabenseifner, R.: Improving the performance of collective operations in MPICH. In: Lecture Notes in Computer Science, Springer (2003) 257–267

[87] Rabenseifner, R., Jesper Larsson Träff: More efficient reduction algorithms for non-power-of-two number of processors in message-passing parallel systems. In: Proceedings of EuroPVM/MPI, Springer (2004) 36–46

[88] Hoefler, T., Lumsdaine, A.: Optimizing non-blocking Collective Operations for InfiniBand. In: Proceedings of the 22nd IEEE International Parallel & Distributed Processing Symposium (IPDPS). (2008)

[89] Gropp, W., Lusk, E., Doss, N., Skjellum, A.: A high-performance, portable implementation of the {MPI} message passing interface standard. Parallel Computing **22**(6) (September 1996) 789–828

[90] Bruck, J., Ho, C.T., Kipnis, S., Weathersby, D.: Efficient algorithms for all-to-all communications in multi-port message-passing systems. Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures (1994) 309

[91] Chen, J., Zhang, Y., Zhang, L., Yuan, W.: Performance Evaluation of Allgather Algorithms On Terascale Linux Cluster with Fast Ethernet. In: High Performance Computing and Grid in Asia Pacific Region, International Conference on. Volume 0., Los Alamitos, CA, USA, IEEE Computer Society (2005) 437–442

[92] Hawkins, D.M.: Identification of Outliers. Monographs on Statistics and Applied Probability. Chapman and Hall, London, New York (1980)

[93] Laboratories, S.N.: Thunderbird Linux Cluster (2009)

[94] Mood, A., Graybill, F., Boes, D.: Introduction to the Theory of Statistics. McGraw-Hill Series in Probability and Statistics. McGraw-hill New York (1974)

[95] Romesburg, C.: Cluster Analysis for Researchers. Lulu.com (2004)

[96] de Hoon, M., Imoto, S., Nolan, J., Miyano, S.: Open source clustering software. Bioinformatics **20**(9) (2004) 1453–1454

[97] Huber, P.J.: Robust statistics. Wiley Series in Probability and Statistics. Wiley-Interscience (February 1981)

[98] Maronna, R.A., Yohai, V.J., Martin, D.R.: Robust Statistics: Theory and Methods. Wiley Series in Probability and Statistics. Wiley, New York (2006)

[99] Rousseeuw, P.J., Leroy, A.M.: Robust Regression and Outlier Detection. John Wiley & Sons, New York (1987)

[100] Lange, K., Little, R., Taylor, J.: Robust statistical modeling using the \$t\$ distribution. Journal of the American Statistical Association **84**(408) (1989) 881–896

[101] Mead, R., Nelder, J.: A simplex method for function minimization. The computer journal **7**(4) (1965) 308

[102] Benkert, K., Gabriel, E., Resch, M.: Outlier Detection in Performance Data of Parallel Applications. In: 9th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing. (2008) 1–8

[103] Gabriel, E., Feki, S., Benkert, K., Resch, M.M.: Towards Performance Portability through Runtime Adaption for High Performance Computing Applications. Concurrency and Computation — Practice and Experience **22**(16) (2010) 2230–2246

Glossary

(n-dimensional) Neighborhood Communication is a communication pattern in which the MPI processes are located in a Cartesian grid and each process communicates with its next neighbors along each of the n axes.

Abstract Data and Communication Library (ADCL) is an auto-tuning library with special focus on tuning communication operations in MPI parallel programs.

Application Programming Interface (API) is a software interface which can be used by other software packages to interact.

Automatic Tuning System encapsulates performance-critical routines (kernels) into a library and offers various codelets for each kernel, timers and search techniques.

Best-performing codelet is the codelet that yields the best performance based on a set of verification runs.

Brute-force Search is a type of search in an empirical optimization library which tests all available codelets.

Codelet is a piece of code that implements one way to solve a particular problem.

Compile-time Tuning is a sub-category of static tuning in which the application program is optimized during compilation.

Computational Fluid Dynamics (CFD) is an area of research which employs numerical methods and algorithms to solve and analyze fluid flows.

Decision Logic is a part of an empirical optimization library which comes up with a decision based on the collected empirical data and determines the optimal codelet, the so-called winner codelet.

Dynamic Tuning see *Runtime Tuning*.

Empirical Optimization is a way of optimizing performance based on empirical data, i.e. measurements, in contrast to model-based optimization strategies.

Exhaustive Search see *Brute-force Search*.

Heuristic is a best-practice rule to rapidly come up with a solution that is close to the optimal solution.

High Performance Computing (HPC) is a discipline that uses supercomputers, i.e. computers at the edge of technology regarding processing capacity.

Install-time Tuning is a sub-category of static tuning in which the tuning library is optimized during installation.

Kernel is a confined entity of a computer program of special importance, e.g. the central part of the algorithm or a computationally expensive well-defined set of operations.

Message-Passing Interface (MPI) is a API specification for parallel computing and very frequently used in case of distributed memory systems.

Network Congestion describes the occurrence of deteriorations in the quality of service (e.g. queueing delay, packet loss) of a network due to excess amounts of data over a certain link.

Optimal codelet see *Winner codelet*.

Outlier is an observation which deviates much from the other observations in the sample.

Runtime Tuning is the opposite of static tuning. It leads to software which has the ability to adapt its behavior at runtime.

Selection Logic is a part of an empirical optimization library which chooses the order in which the codelets are evaluated during the search.

Static Tuning is the process of optimizing a code sequence/application a priori of the actual execution of the program resulting in software that cannot alter its behavior during execution. Static tuning is sometimes sub-classified into install-time and compile-time tuning.

Verification runs denotes a set of measurements where each codelet in a set of codelets is preselected and executed once in a long-term run.

Winner codelet is the codelet the decision logic judges best-performing and which is subsequently used during the production phase.

CV

Personal Data

Name	Katharina Benkert
born	22 July 1978 in Augsburg
family status	Married

Education

09/1990–07/1997	Justus-von-Liebig Gymnasium Neusäß A-levels
10/1997–09/1998	Fern-Universität Hagen
10/1998–07/2004	TU Chemnitz Diploma: Applied Mathematics (Dipl.-Math. techn.) Focus: Numerics
09/2000-07/2001	Université de Toulouse le Mirail Diploma: Applied Mathematics (Licence)

Profession

10/2004-03/2011	Research assistant at the High Performance Computing Center Stuttgart (HLRS)

Stuttgart, February 7, 2011