Universität Hamburg

DER FORSCHUNG | DER LEHRE | DER BILDUNG

D6.2 Collection of Success Stories

Kai Himstedt, Nathanael Hübbe, Julian Kunkel, Sandra Schröder, and Hinnerk Stüben

Work Package:	WP6
Responsible Institution:	RRZ, DKRZ
Date of Submission:	December 2019

Contents

1 Introduction **4**

2 Success Stories **5**

3 Success Stories based on Code Co-Development and Tuning **6**
3.1 Statistics Package R: Regression Analysis 6
3.1.1 Problem Description 6
3.1.2 Procedure 6
3.1.3 Results 6
3.1.4 Applied Performance and Software Engineering Concepts 7
3.1.5 Material 7
3.2 Statistics Package R: Analyzing Satellite Night Images 7
3.2.1 Problem Description 7
3.2.2 Procedure 8
3.2.3 Results 8
3.2.4 Applied Performance and Software Engineering Concepts 8
3.2.5 Material 9

4 Success Stories based on Code Co-Development **10**
4.1 Teaching Software Engineering Concepts 10
4.1.1 Problem Description 10
4.1.2 Procedure 10
4.1.3 Results 10
4.1.4 Applied Performance and Software Engineering Concepts 11
4.1.5 Material 11
4.2 Finding an Insidious Bug in a Large Fortran Program 12
4.2.1 Problem Description 12
4.2.2 Procedure 12
4.2.3 Results 13
4.2.4 Applied Performance and Software Engineering Concepts 13
4.2.5 Material 13

5 Success Stories based on Tuning **14**
5.1 Statistics Package R: Using Efficient Libraries 14
5.1.1 Problem Description 14
5.1.2 Procedure 14
5.1.3 Results 14
5.1.4 Applied Performance and Software Engineering Concepts 14
5.1.5 Material 15
5.2 Parallelization of Decompression in CDI 15

5.2.1 Problem Description 15
5.2.2 Procedure 15
5.2.3 Results 20
5.2.4 Applied Performance and Software Engineering Concepts 20
5.2.5 Material 20
5.3 Automatic Tuning Using a Black Box Optimizer Tool 21
5.3.1 Problem Description 21
5.3.2 Procedure 21
5.3.3 Results 22
5.3.4 Applied Performance and Software Engineering Concepts 23
5.3.5 Material 23

Chapter 1

Introduction

As part of the PeCoH project, standard software as well as individual software were enhanced and tuned. Success stories from different scientific fields shall demonstrate the benefit of performance and software engineering. For this purpose we cooperated with scientists and attended to them throughout the project period. Several success stories are closely related to code co-development. By publishing these stories we try to reach a better acceptance of performance and software engineering in the field of HPC in order to increase the productivity of scientists. Together with scientists, we established pilot studies to support re-write of existing codes and to tune parallel programs. The results are documented as success stories. The pilot studies are selected to represent exemplary applications in order to make the results transferable in generalised form to similar problems. This enables the derivation of best practices and strategies and it supports the scientists in their daily work.

The collection of the success stories includes studies on topics such as

- encouraging HPC users who have only used simple editors and the command line interface for the program development to consider integrated development environments (IDEs) like Eclipse [Ecl19] and Visual Studio Code (VS Code) [Mic18]
- teaching important software engineering concepts like refactoring, consistent coding style, documentation, debugging, and unit testing based on a tutorial carried out by HPC users
- finding insidious bugs in large Fortran programs using tools
- parallelising R programs using the foreach() parallelization paradigm
- achieving performance improvements for R programs by using efficient libraries like OpenBLAS or MKL, an appropriate compiler and MPI environment, and appropriate compiler options
- parallelising the reading of GRIB (GRIdded Binary or General Regularly-distributed Information in Binary form) data using the Climate Data Interface (CDI) [MPI19a]
- automatically finding the parameter combinations for building and running parallel applications that give the best benchmark results using a Black Box Optimizer Tool, which is based on genetic algorithms. Build parameters include the selection of the best performing compiler, MPI environment, and runtime parameters relate to the best setting of MPI options and application specific options for an appropriate partitioning, tiling, ...

Chapter 2

Success Stories

Three types of success stories are presented below, distinguished according to whether the focus is on a) code co-development and tuning, b) code co-development, or c) tuning. To make the success stories potentially suitable as input for a knowledge base, we designed and applied a *generic template* for the outline of a success story.

The template is divided into five sections:

1. *problem description* characterizing the initial situation.
2. *procedure* outlining the concrete steps and the methods used.
3. *results* emphasizing the advantages and disadvantages of the applied concepts.
4. *software and performance engineering concepts* that have been used for the code co-development and/or tuning. For detailed information about the performance and software engineering concepts that are used below refer to [HHKS18].
5. *material* that have been used in the code co-development process.

Chapter 3

Success Stories based on Code Co-Development and Tuning

The following optimizations have been co-developed with scientists based on their observation that the code performs suboptimal.

3.1 Statistics Package R: Regression Analysis

3.1.1 Problem Description

The runtimes of an R program using the rlassoEffects-function [SCH18], were too high for larger problem sizes.

3.1.2 Procedure

Together with the HPC user we reproduced the problem by deriving benchmarks from the problem. Next, we explained possible ways to parallelize the program in order to improve its performance. In this case a parallelization was implemented by replacing sequential loops with parallel loops using existing R packages like doMPI, foreach, iterators, and Rmpi [Wes17].

Benchmarks were performed for a small and a larger problem size using up to 8 cluster nodes and up to 16 physical/32 hyper-threaded cores on each node. To avoid measuring inaccuracies and to avoid effects caused by special features of current CPU architectures, the problem sizes are chosen in a way that runtimes are several seconds at least for the small problem size and half a minute at least for the larger problem size. For instance, the CPU clock rates of a typical multi-core cluster node supporting features like turbo boost may vary depending on the CPU usage. At low CPU load, for example, if only a single core is used, the clock rate of this core is typically considerably higher than the clock rate at times when several cores are fully utilized. If many cores are fully utilized over a period of time, the clock rates of the cores will usually be reduced over time to avoid a rise in CPU temperature, which is determined by the clock rates.

3.1.3 Results

It was observed that one additional core of the first node is needed by the R MPI runtime environment for internal purposes when a benchmark is performed via `mpirun`. For the sake of simplicity, this additional core is neglected in the calculation of speedup and efficiency. Except for the case where only one node was used for the bigger problem

size, the results showed that hyper-threading has a negative impact on the speedups. In general the speedups achieved are better for the bigger problem size. For the small problem size a speedup of 9.36 is achieved on two cluster nodes, each using 16 cores (efficiency 30.18%). No meaningful speedup can be achieved using four or more nodes. For the bigger problem size, we achieved a speedup of about 30 on four cluster nodes, each using 16 cores (efficiency 44.78%). With 8 nodes hardly any improvement is achieved. For detailed information about the benchmark results refer to Section “Three use Cases for R Programs” – Use Case B in Deliverable 5.1 [Him19].

3.1.4 Applied Performance and Software Engineering Concepts

Listed below are the performance and software engineering concepts which were applied within the scope of this success story. For more information about these concepts refer to [HHKS18].

PE2.1 Using Standard Tools to Measure System Performance
Notes: elapsed runtime of a program measured

PE3 Benchmarking
Notes: controlled experiments performed to measure speedups and efficiencies by providing varying HPC resources, i.e., 1, 2, 4, 8, ... nodes on a distributed system

PE4.3 Tuning via Reprogramming
Notes: parallel outer loop added

SE1.2.1 Parallel Algorithms
Notes: embarrassingly (i.e., trivially) parallelizable algorithm

SE1.2.3 Programming Message Passing Systems
Notes: MPI used as the de-facto standard for parallelizing a program in distributed environments like HPC cluster systems

SE1.2.4 Load Balancing
Notes: simple scheduling, as a result from foreach() paradigm, achieves an appropriate distribution of the workloads across the multiple computing resources of the HPC system

SE2.3 Programming Idioms
Notes: foreach() paradigm

3.1.5 Material

The benchmarks were based on test data provided by the HPC user with whom we have jointly performed the code co-development.

3.2 Statistics Package R: Analyzing Satellite Night Images

3.2.1 Problem Description

The runtimes of an R program for analyzing satellite night images were too high for practical use.

3.2.2 Procedure

Together with the HPC user we reproduced the problem by the help of benchmarks. Next we explained possible ways to parallelize the program in order to improve its performance. In this case a parallelization was implemented by replacing sequential loops with parallel loops using appropriate R packages like doMPI, foreach, iterators, and Rmpi [Wes17].

Benchmarks were performed with up to 32 cluster nodes and up to 16 physical/32 hyper-threaded cores on each node. A challenge was the large demand of the program for main memory.

3.2.3 Results

The R runtime environment uses a workspace that includes all user-defined objects (vectors, matrices, lists, functions, ...). In connection with the R MPI package it was observed that for each parallel process, this workspace is replicated. On nodes with many cores this may lead to an out-of-memory problem.

One idea to avoid this problem is not to use all cores of a node. It must be assessed in the individual case whether the underutilization of nodes (but instead using more nodes) seems appropriate to further reduce the time to solution. On 32 cluster nodes, each using 4 cores, we achieved a speedup of 126 compared to sequential run. For detailed information about the benchmark results refer to Section “Three use Cases for R Programs” – Use Case C in Deliverable 5.1 [Him19].

3.2.4 Applied Performance and Software Engineering Concepts

Listed below are the performance and software engineering concepts which were applied within the scope of this success story. For more information about these concepts refer to [HHKS18].

PE1 Cost Awareness

Notes: the underutilization of nodes was accepted considering the time to solution constraints

PE2.1 Using Standard Tools to Measure System Performance

Notes: elapsed runtime of a program measured

PE3 Benchmarking

Notes: controlled experiments performed to measure speedups and efficiencies by providing varying HPC resources, i.e., 1, 2, 4, 8, ... nodes on a distributed system

PE4.3 Tuning via Reprogramming

Notes: parallel outer loop added

SE1.2.1 Parallel Algorithms

Notes: embarrassingly (i.e., trivially) parallelizable algorithm

SE1.2.3 Programming Message Passing Systems

Notes: MPI used as the de-facto standard for parallelizing a program in distributed environments like HPC cluster systems

SE1.2.4 Load Balancing

Notes: simple scheduling, as a result from foreach() paradigm, achieves an appropriate distribution of the workloads across the multiple computing resources of the HPC system

SE2.3 Programming Idioms

Notes: foreach() paradigm

3.2.5 Material

The benchmarks were based on test data provided by the HPC user with whom we have jointly performed the code co-development.

Chapter 4

Success Stories based on Code Co-Development

4.1 Teaching Software Engineering Concepts

4.1.1 Problem Description

Scientists often use text editors like `vim`[ubu19b], `emacs`[GNU19], or `nano`[ubu19a] to write the source code. While those editors support at least source code highlighting, they lack advanced features like debugging, automatic refactoring, code structure views, or formatting support, to name just a few. That is why the Eclipse Integrated Development Environment (IDE) has been chosen as an exemplary IDE to teach such important software engineering concepts and to explore the benefit an IDE may have on scientists' workflow.

4.1.2 Procedure

To keep the code co-development as simple as possible for HPC developers but with a good transferability of results, we have chosen the use of an Integrated Development Environment (IDE), refactoring, consistent coding style, documentation, debugging, and unit testing as particularly relevant concepts, given the experience that many HPC users are not aware of the benefits of applying these concepts in practice. Then we asked scientists to apply them in their everyday work. We designed a tutorial to teach them the most important principles of the software engineering practices. In order to collect the experiences of the code co-development process, the participant needed to fill out a survey. Additionally, we interviewed the participants.

4.1.3 Results

The results are based on the feedback mentioned above and collected by talking to the participants and meets our expectations: Indentation and naming conventions helped to enhance and keep the understandability of the code and introducing naming convention does not require high effort. Refactoring to divide the code into functions of shorter length keeps the structure of the code and improves understandability. Documentation in form of code comments improves understandability of code.

However, during the code co-development process we found that it is challenging to introduce software development methods into the development process of scientists in the context of HPC. Especially the documentation of the code was considered time

consuming. Scientists fear that following software engineering practices might slow down the entire research process. One plausible explanation for this is that they mostly write code without having sustainability in mind and without considering that the code will often be shared with other scientists. But even if the code will not be shared with other scientists, it can be assumed that a break-even will be reached when scientists use modern software development methods.

Nevertheless, further studies should be carried out to broaden the use of software engineering techniques in the field of HPC in order to increase the performance of parallel programs.

For a detailed description of the experience report on the code co-development process refer to Deliverable 2.2 [Sch19].

4.1.4 Applied Performance and Software Engineering Concepts

Listed below are the performance and software engineering concepts which were applied within the scope of this success story. For more information about these concepts refer to [HHKS18].

SE2.1 Integrated Development Environments
Notes: configuration and usage of the integrated development environment (IDE) Eclipse, e.g., to seamlessly perform the typical development cycle with the steps edit, build (compile and link), and test

SE2.2 Debugging
Notes: using a sophisticated debugger via the IDE to perform the common debugging workflow based on commands like step into, step over, step out, set breakpoint as a very helpful and supportive method to find and resolve defects within a program

SE4.1 Test-driven Development and Agile Testing
Notes: unit testing ensures that a part of an application – the unit – meets its requirements, i.e., that it behaves as intended

SE5.1 Coding Standards
Notes: consistent coding style, e.g., regarding indentation and naming conventions, enhances the understandability of the source code

SE5.3 Refactoring
Notes: applying common code refactoring improves code quality

SE7.3 Source Code Documentation
Notes: code comments improve the understandability of code

4.1.5 Material

As supplementary material a tutorial has been designed for the participants for each selected concept (also refer to Deliverable 2.2 [Sch19]).

4.2 Finding an Insidious Bug in a Large Fortran Program

4.2.1 Problem Description

The BQCD (Berlin quantum chromodynamics program) [ABS08] written in Fortran did not utilize SIMD instructions. As part of this case study, BQCD was ported to utilize SIMD instructions; also, appropriate macros for various operations with complex numbers were implemented. After several test runs, the results of the ported BQCD version, hereinafter named $BQCD_{SIMD}$, were identical with the results of the original version, hereinafter named $BQCD_{REF}$. Therefore, the new SIMD functionality was considered to be successfully tested. But after further testing, it surprisingly became clear that $BQCD_{SIMD}$ produces incorrect results for certain input data. A direct relation with the implementation of the SIMD based macros was therefore not immediately obvious. As a lucky circumstance for finding the bug it can be noted that $BQCD_{REF}$ and $BQCD_{SIMD}$ each behave in a deterministic way for the same input.

4.2.2 Procedure

To fix the bug, a pragmatic approach was chosen: both BQCD versions were run simultaneously using two separate debuggers belonging to their respective Visual Studio Code (VS Code) [Mic18] IDE, while the same input data was provided. For managing both debug environments, two virtual machines (VMs) were configured on a single physical PC, for which differences arise only from the differences in the source code of both BQCD versions. The PC had two displays connected to it so that both code version could be debugged on one monitor in fullscreen mode, i.e., $BQCD_{SIMD}$ on one, and $BQCD_{REF}$ on the other.

For debugging the following approach was taken:

Initially the total number of iterations imax for the outermost loop of $BQCD_{REF}$ is determined in a pretest. Then the debugging ist started simultaneously for both versions setting breakpoints in a way that both versions are interrupted after the first half of the total number of iterations $imax/2$. At the time of the interruption of both programs, all relevant data structures which may influence the differences in the final results after having completed all iterations, are examined with the corresponding debugger and their contents are compared between both environments. Two cases are to be distinguished: a) the contents of the relevant data structures already diverge, indicating that an incorrect calculation must have been carried out before, or b) the contents are still identical, indicating that an incorrect calculation will be carried out in a later iteration.

In the case of a) the debugging for both BQCD versions is restarted and this time the breakpoints are set in a way that both versions are interrupted after the first quarter of the total number of iterations $imax/4$. In the case of b) the breakpoints are set in a way that both versions are interrupted after three-quarters of the total number of iterations $imax \cdot 3/4$. In analogy to a bisection method, this process repeats until the loop index i_{error} is determined, where the inaccurate calculation in $BQCD_{SIMD}$ is first discovered.

After narrowing the error in this way, the debugging is restarted and the breakpoints are set to interrupt the execution at loop index $i_{error} - 1$. Via single stepping from this point synchronously through both versions the error can be further narrowed by examining the relevant data structures and comparing their contents between both environments after each step. In the present case, this was sufficient to identify the macro that was responsible for the faulty calculation. The error then became immediately apparent and the bug was fixed using an additional temporary variable for a certain arithmetic

operation with complex numbers. If single stepping through the programs seems to be too time consuming, the foregoing procedure can be used in an analogous way, e.g., by applying it to inner loops relative to loop index $i_{error} - 1$.

4.2.3 Results

Using IDEs and applying a schematic approach to use debuggers an insidious bug in a large Fortran program could be found in about half an hour (not counting the time for initially setting up the IDEs and debugging environments). The error was originally introduced by porting a macro to tune a calculation with complex numbers by the help of SIMD instructions: just missing was the buffering of the actual content of a variable in a temporary variable in order to be able to use this content after it gets overwritten (similar to the idiom used for a typical swap operation).

The developer of the macro additionally used an alternative approach to find the error by the help of inserting print statements into the source code at “suspicious locations”. In this way the error could also be found very fast. Inserting print statements into source code to find errors is still widely used, especially in the field of HPC. In contrast to the schematic approach, however, this alternative requires expert knowledge and a great familiarity with the corresponding source code to make this approach, which is more concerned with intuition, efficient. In connection with the procedure using two debuggers in the way described above the question arises if it could be automated or at least largely automated for analogous cases, where – in a deterministic way – a reference version of a program produces correct results and a modified version of the same program produces inaccurate results for certain input data.

4.2.4 Applied Performance and Software Engineering Concepts

Listed below are the performance and software engineering concepts which were applied within the scope of this success story. For more information about these concepts refer to [HHKS18].

SE2.1 Integrated Development Environments

Notes: configuration and usage of the integrated development environment (IDE) Visual Studio Code (VS Code) [Mic18], e.g., to seamlessly perform the typical development cycle with the steps edit, build (compile and link), and test

SE2.2 Debugging

Notes: using a sophisticated debugger via the IDE to perform the common debugging workflow based on commands like step into, step over, step out, set breakpoint as a very helpful an supportive method to find and resolve defects within a program

4.2.5 Material

The development of the BQCD (Berlin Quantum Chromodynamics) program was started in 1998 by Stüben for the two flavour case and the original Wilson action [ABS18]. The sources are available for download [ABS18]. Here we used the sources of an unmodified BQCD version and the sources of a SIMD ported version.

Chapter 5

Success Stories based on Tuning

5.1 Statistics Package R: Using Efficient Libraries

5.1.1 Problem Description

Many urgent needs for tuning programs written in the language R were brought to us by users.

5.1.2 Procedure

Benchmark experiments were performed based on the R Benchmark 2.5 test suite [Urb18] and building and/or selecting optimized Mathlibs (i.e., OpenBLAS or MKL). The test suite contains three sections named “Matrix calculation”, “Matrix functions”, and “Programmation” containing 5 tests each, giving 15 tests in total. Each test is run three times to obtain more accurate results. The test suite consists mainly of a mix of matrix operations (e.g., cross product, eigenvalues, ...) and algorithmic parts (e.g., recursion, loops, ...). Additional experiments were performed using the environment variable OMP_NUM_THREADS in order to exploit parallelism using several threads.

5.1.3 Results

The experiments showed that the selection of an efficient library like OpenBLAS or MKL leads to good performance improvements with respect to the use of the standard library. In the experiments where binaries were built from source, an optimization level of `-O3` (Intel compiler) gave the best results, whereas using Profile Guided Optimization (PGO) was not beneficial. The speedup results for MKL where usually better than those achieved with OpenBLAS by a small margin. Using one core on a single cluster node, we achieved a speedup of about 5 for the configuration with the MKL library. Hardly any additional speedup could be achieved for the experiments were up to 16 physical threads were used setting the environment variable OMP_NUM_THREADS in order to exploit parallelism on a single cluster node (about 15% (OpenBLAS) and about 18% (MKL) compared to a single core). For detailed information about the benchmark results refer to Section “Three use Cases for R Programs” – Use Case A in Deliverable 5.1 [Him19].

5.1.4 Applied Performance and Software Engineering Concepts

Listed below are the performance and software engineering concepts which were applied within the scope of this success story. For more information about these concepts refer to [HHKS18].

PE2.1 Using Standard Tools to Measure System Performance
Notes: elapsed runtime of a program measured

PE3 Benchmarking
Notes: controlled experiments performed to measure speedups and efficiencies by providing varying resources of a single cluster node, i.e., 1, 2, 4, 8, ... cores on a shared memory system

PE5 Optimization Cycle
Notes: various parameter combinations were manually examined via the cycle *set next parameter combination* $\rightarrow$ *benchmark*

5.1.5 Material

The benchmarks were based on the R Benchmark 2.5 test suite [Urb18].

5.2 Parallelization of Decompression in CDI

5.2.1 Problem Description

The Climate Data Interface (CDI) [MPI19a] is a library that is used for I/O in many programs used by climate scientists. This library provides a unified interface to read and write data in different relevant file formats like GRIB (GRIdded Binary or General Regularly-distributed Information in Binary form) and NetCDF (Network Common Data Form) [Uni19]. Some of these formats have the feature that they may contain compressed data, which needs to be decompressed when read. However, it was observed that read performance dropped by a factor of 5 when reading compressed data, even though less data needs to be fetched from storage.

5.2.2 Procedure

First, to analyze the situation, the involved libraries were instrumented to measure the performance systematically. Then, the optimization of the overall workflow was performed by implementing a non-invasive parallelization scheme.

We were provided with access to the current source codes of both CDI and CDO, as well as a set of large GRIB files that used AEC compression. The simple program app/cdi was employed as a test. This program, which is included with the source code of the CDI library, uses the CDI library to both read a file, and then store the same data in a new file. In addition to the CDI library, the libaec was compiled from source (which is provided by the German Climate Computing Center). This allowed for instrumentation of both the high-level calls within the CDI library, as well as the low-level calls into the libaec. Between these two layers is the GribAPI library, which is quite difficult to compile from source. The GribAPI is used by CDI to decode the GRIB messages within a GRIB file, and it is the GribAPI which then uses libaec to perform the (de-)compression of the data. Instrumentation of CDI itself was straight-forward, instrumentation of libaec required use of the LD_PRELOAD environment variable to override the libaec version referenced by the standard GribAPI installation on the mistral.

After instrumenting the involved code and measuring its performance in detail, we concluded that CDI itself was not to blame. Instead, the libraries that are used by CDI are responsible for the slowdown. Since the more important of these two libraries is

rather hard to change, we decided to simply parallelize the entire decoding process. In our parallelization, the application's main process reads the compressed chunks of data, dispatches them to worker processes for decoding, and collects the results for passing them back to the user of CDI. This parallelization was implemented in a way that makes it transparent to the user code.

Instrumentation

Sequential Version Instrumentation recorded the following wall-clock times, together with their respective payload data amount (compression is associated with data size *before* compression and *after* decompression):

- Read side:
  - read:
    Time to fetch a GRIB record from storage.
    The data is not interpreted in any way.
  - unzip:
    Decompression of data that is initiated by CDI directly.
    This seems not to be used within the test setup.
  - decode:
    Interpretation of the record's data using the GribAPI to retrieve the payload data contained within.
    This may include a decompression step that's initiated by the GribAPI.
  - AEC:
    Time spent within libaec.
    This is a part of the decode time.
- Write side:
  - encode:
    Reverse of decode, this takes the payload data and turns it into a GRIB record.
  - zip:
    Compression that is initiated by CDI directly.
    Not used in the test setup.
  - write:
    Output the in-memory GRIB record to disk.

All instrumentation captures the times for each call of the respective subroutine, updates a static accumulator variable, and outputs the current value of the accumulator. That way, the last output will always give the times aggregated over all calls.

Parallelized Version To verify the success of the parallelization, the resulting code was instrumented again, providing the following times:

- Read side:

  - read:
    As with the sequential code, the time used to actually read data from disk.
    Same as before parallelization.
  - request:
    The time spent by the main thread to dispatch work to the worker threads.
  - wait:
    The time spent by the main thread waiting for an async job to finish.
  - copy:
    The time spent by the main thread copying the worker's results back into the user supplied buffer.
  - async decode:
    The time spent by the *worker threads* to decode the data.
    This is expected to be a multiple of the total read time.
  - sync decode:
    Any time that the main thread spends decoding data itself because the requested data could not be fetched from a worker.

- Write side:

  - encode:
    Same as before parallelization.
  - zip:
    Same as before parallelization.
  - write:
    Same as before parallelization.

Test Setup

For our tests, four different files were selected from the given set of files, and from each file the first timestep was extracted using CDO. This served to cut down on execution time requirements for the tests, as the original files have sizes in the range between 226 MB and 69 GB. With the one-timestep reduction, the four test files had the following sizes:

```
var1-compr.grb:  236175040 bytes
var2-compr.grb: 7308019640 bytes
var3-compr.grb:  700737416 bytes
var4-compr.grb: 4516496832 bytes
```

Also, copies of all four files were created which do not use AEC compression by executing `app/cdi varX-{,un}compressed.grb`, the resulting uncompressed files had the following sizes:

```
var1-uncompr.grb:   838861510 bytes
var2-uncompr.grb: 12918469102 bytes
```

```
var3-uncompr.grb:  1677723020 bytes

var4-uncompr.grb: 12918469102 bytes
```

The CDI version tested is (git commit 6130d637d3f2bb0c6dad3b8e9613063fcf281340), the libaec version 1.0.4 (commit 0c0453a0e463da9c2183f46d0255f05645e0e5ef).

Measurements

Sequential Version

	file 1	file 2	file 3
input	3355MB/8.569s	51674MB/154.712s	6711MB/20.163s
read	236MB/0.088s	7308MB/ 2.452s	701MB/ 0.226s
unzip	236MB/0.000s	7308MB/ 0.000s	701MB/ 0.000s
decode	3355MB/8.481s	51674MB/152.261s	6711MB/19.937s
AEC	839MB/3.945s	12918MB/ 85.311s	1678MB/10.975s
output	3355MB/5.209s	51674MB/76.302s	6711MB/9.949s
encode	3355MB/3.756s	51674MB/52.639s	6711MB/7.060s
zip	839MB/0.000s	12918MB/ 0.000s	1678MB/0.000s
write	839MB/1.453s	12918MB/23.663s	1678MB/2.889s

	file 4	total
input	51674MB/150.474s	113GB/334s = 340MB/s
read	4516MB/ 1.761s	13GB/ 5s = 2819MB/s
unzip	4516MB/ 0.000s	13GB/ 0s
decode	51674MB/148.713s	113GB/329s = 344MB/s
AEC	12918MB/ 82.728s	28GB/183s = 155MB/s
output	51674MB/74.860s	113GB/166s = 682MB/s
encode	51674MB/51.402s	113GB/115s = 987MB/s
zip	12918MB/ 0.000s	28GB/ 0s
write	12918MB/23.458s	28GB/ 51s = 551MB/s

Table 5.1: Compressed Input

	file 1	file 2	file 3
input	3355MB/1.835s	51674MB/23.033s	6711MB/4.575s
read	839MB/0.346s	12918MB/ 5.035s	1678MB/1.949s
unzip	839MB/0.000s	12918MB/ 0.035s	1678MB/0.000s
decode	3355MB/1.489s	51674MB/17.962s	6711MB/2.626s
AEC			
output	3355MB/5.251s	51674MB/77.027s	6711MB/9.877s
encode	3355MB/3.789s	51674MB/53.185s	6711MB/7.010s
zip	839MB/0.000s	12918MB/ 0.000s	1678MB/0.000s
write	839MB/1.462s	12918MB/23.843s	1678MB/2.867s

	file 4	total
input	51674MB/36.875s	113GB/ 66s = 1710MB/s
read	12918MB/18.135s	28GB/ 25s = 1113MB/s
unzip	12918MB/ 0.000s	28GB/ 0s
decode	51674MB/18.740s	113GB/ 41s = 2779MB/s
AEC		
output	51674MB/76.852s	113GB/169s = 671MB/s
encode	51674MB/52.484s	113GB/116s = 974MB/s
zip	12918MB/ 0.000s	28GB/ 0s
write	12918MB/24.367s	28GB/ 53s = 540MB/s

Table 5.2: Uncompressed Input

Parallelized Version, 8 Worker Threads

	file 1	file 2	file 3
input	4.082s	30.819s	5.331s
read	0.240s	15.692s	1.298s
request	0.000s	0.000s	0.000s
wait	2.760s	1.439s	1.909s
copy	1.222s	15.620s	2.253s
sync	0.000s	0.000s	0.000s
async	11.750s	213.933s	27.788s
output	5.538s	82.287s	10.519s
encode	3355MB/3.936s	51674MB/56.527s	6711MB/7.274s
zip	839MB/0.000s	12918MB/0.000s	1678MB/0.000s
write	839MB/1.602s	12918MB/25.760s	1678MB/3.245s

	file 4	total
input	25.925s	66s
read	9.932s	27s
request	0.003s	0s
wait	2.162s	8s
copy	13.639s	33s
sync	0.000s	0s
async	208.540s	462s
output	83.926s	182s
encode	51674MB/57.669s	113414MB/125s = 904.374591MB/s
zip	12918MB/0.000s	28353MB/0s
write	12918MB/26.257s	28353MB/57s = 498.61072MB/s

Table 5.3: Compressed Input

	file 1	file 2	file 3
input	2.430s	39.282s	5.949s
read	1.168s	26.131s	3.570s
request	0.000s	0.003s	0.000s
wait	0.000s	0.000s	0.000s
copy	1.302s	14.808s	2.366s
sync	0.000s	0.000s	0.000s
async	4.372s	63.205s	8.910s
output	5.558s	81.112s	11.067s
encode	3355MB/3.910s	51674MB/55.674s	6711MB/7.803s
zip	839MB/0.000s	12918MB/ 0.000s	1678MB/0.000s
write	839MB/1.649s	12918MB/25.438s	1678MB/3.264s

	file 4	total
input	41.599s	89s
read	26.256s	57s
request	0.000s	0s
wait	0.000s	0s
copy	14.641s	33s
sync	0.000s	0s
async	64.493s	141s
output	84.060s	182s
encode	51674MB/58.155s	113414MB/126s = 903.39488MB/s
zip	12918MB/ 0.000s	28353MB/0s
write	12918MB/25.905s	28353MB/56s = 503.999573MB/s

Table 5.4: Uncompressed Input

5.2.3 Results

The parallelization succeeded in accelerating input to the former speed of reading uncompressed data (5x speedup). However, reading uncompressed data with parallelization switched on is a bit slower than serial processing (factor 1.35) due to the additional movement of data between master and worker processes. Of course, it is possible switch off parallelization in that case, making reading of compressed and uncompressed data equally fast.

It was critical to perform the measurements first for this optimization effort, as the resulting numbers quickly allowed us to rule out a number of approaches as not efficient enough. As a consequence, we could settle on the best solution right away.

The users were satisfied with the outcome of this co-development project: *"The acceleration is quite visible and works as expected. It is even beneficial for weakly compressed GRIB files if low resolutions are used."* says Luis Kornblueh of the Max Planck Institute for Meteorology.

5.2.4 Applied Performance and Software Engineering Concepts

Listed below are the performance and software engineering concepts which were applied within the scope of this success story. For more information about these concepts refer to [HHKS18].

PE2.1 Using Standard Tools to Measure System Performance
Notes: instrumentation was used to record wall-clock times, together with their respective payload data amount (compression is associated with data size *before* compression and *after* decompression)

PE3 Benchmarking
Notes: controlled experiments were performed to measure I/O performance for file operations like read, unzip, decode, AEC compression, encode, zip, and write

PE4.3 Tuning via Reprogramming
Notes: dispatching chunks of data to parallel worker processes

SE1.2.1 Parallel Algorithms
Notes: the application's main process reads the compressed chunks of data, dispatches them to worker processes for decoding, and collects the results for passing them back to the user of CDI

SE1.2.4 Load Balancing
Notes: simple scheduling by dispatching chunks of data to parallel worker processes

5.2.5 Material

We were provided with access to the current source codes of both CDI and CDO, as well as a set of large GRIB files that used AEC compression.

5.3 Automatic Tuning Using a Black Box Optimizer Tool

5.3.1 Problem Description

In the beginning of the project, we started with manually tuning programs. The typical tuning workflow was represented by an optimization cycle with the steps *set parameter combination*, *benchmark the program*, and *evaluate benchmark result to determine the next improved parameter combination*. This approach was suitable to produce good tuning results, but traditional manual tuning was more time consuming than anticipated. For this reason, we realigned the focus to perform experiments using a Black Box Optimizer Tool based on genetic algorithms, in order to automatically find the parameter combination for a parallel application that gives the best benchmark result.

5.3.2 Procedure

In initial experiments the Black Box Optimizer was used to automatically tune two small MPI test programs: a parallel program to calculate π and a parallel solver for boolean satisfiability problems (SAT). The experience from these experiments provided the basis for tuning two real applications afterwards with the Black Box Optimizer: BQCD (Berlin Quantum Chromodynamics program) [ABS18], a parallel program for simulating lattice QCD with dynamical Wilson fermions, and Fesom2 (Finite-Element/volumE Sea ice-Ocean Model) [Fes19], a parallel program for simulating the circulation of the global ocean with regional focus.

Automatically tuning these parallel programs was performed without modifying the source code, e.g., by setting appropriate runtime options and selecting the best performing compiler (GNU, Intel, or PGI) and MPI environment (Intel MPI or Open-MPI) for each specific program. For a small number of parameters, it would have been possible to benchmark all parameter combinations in the sense of a brute-force approach. But with the increase of parameters, the search space increases exponentially and a brute-force is no longer suitable. Aside from choosing the most appropriate compiler (GNU, Intel, PGI, ...) in the most suitable version (latest version, second latest version, ...) and in the most appropriate combination with an MPI environment (Intel MPI, Open-MPI, ...) there are many other runtime options, for example for the selection of the algorithm for collective MPI operations (e.g., Recursive doubling, Rabenseifner's, Reduce Bcast, ... for MPI_Allreduce), for process pinning and mapping, or for the use of hyper-threading. In addition, there were application specific options for BQCD, e.g., for partitioning and tiling. Genetic algorithms are widely used in practice as a strategy to reduce such huge search spaces efficiently.

Genetic algorithms are an appropriate method to solve complex optimization problems which are not solvable with analytical methods. A specific combination of parameters (known as *genes*) is named an *individual* or *chromosome*. The basic idea is to select and combine two good solutions to produce an even better solution by applying the genetic operations *crossover* (i.e., exchanging genes of the parents) and *mutation* (i.e. slightly modifying a gene with a low random probability). The optimization process starts with a set of (typically randomly created) individuals (known as the *population*) which is named initial *solution*. New solutions are produced until a sufficiently good solution has been found or the solution cannot be further improved.

5.3.3 Results

For the parallel test program to calculate π we selected 6 input parameters for tuning: CompilerSelection (GCC, Intel, ...), CompilerGeneration (latest, second latest, ...), MPISelection (Open-MPI, Intel MPI), OptimizationLevel (-O0, -O1, -O2, -O3, ...), UseProfileGuidedOptimization (on, off), and UseHyperthreading (on, off). The execution of the parallel π program was configured to use 4 cluster nodes, each with 16 physical cores. For a brute-force approach 480 benchmarks would be required to analyse all possible combinations of the 6 input parameters. The Black Box Optimizer found the best solution after 100 runs in total. Two interesting observations were that the latest compiler generation is not always the fastest and that hyper-threading and Profile Guided Optimization (PGO) are sometimes helpful. For purposes of comparison, we tuned the π application manually as well but we were not able to achieve a better result than the Black Box Optimizer.

While the π calculation was characterized by floating point calculations for the second parallel test program, we chose a simple parallel solver for boolean satisfiability problems (SAT) from the artificial intelligence (AI) domain. The six input parameters to optimize were the same as for the π calculation and the execution of the solver was configured to use four cluster nodes, each with 16 physical cores. This time the Black Box Optimizer found the best solution after a total of 60 benchmarks.

For BQCD (first real-world program) a suitable partitioning, i.e., the best mapping of work to processing elements, is of central importance for the performance. In contrast to the first two experiments the build step is omitted. For BQCD we selected 8 input parameters for tuning: three partitioning parameters, three internal BQCD parameters, a parameter to control hyper-threading, and a parameter to control SIMD execution. For the sake of simplicity, we do not go into more detail about the meaning of these parameters. It is not necessary to know their meaning, because also the Black Box Optimizer modifies these parameters without any knowledge about their meaning. For a brute-force approach, 20736 benchmark runs would be required to analyse all possible combinations of the 8 input parameters. The execution of BQCD was configured to use 8 cluster nodes, each with 16 physical cores. The best solution was found after 900 benchmarks, which was about 10–15% faster than results achieved with parameter settings based on educated guesses by the author of the BQCD program.

For Fesom2 (second real-world program), similar to optimization of the π and SAT programs a build step is involved to determine the best compiler, best compiler generation, and so on. Fesom2 can be regarded as a typical MPI program for which load imbalances may play a role caused by the lack of structure of the grids. Two Fesom2 experiments were carried out: For the first experiment the MPI parameters were tuned manually in advance and for the second experiment they were additionally tuned by the Black Box Optimizer. For the first Fesom2 experiment, we selected 9 input parameters for tuning, the first 6 parameters being the same as for tuning π and SAT. The three additional parameters are: MathlibSelection (MKL, OpenBLAS), ProcessBindingSelection (bind process to core, bind process to hwthread, bind process to socket, ...), and ProcessMappingSelection (blocked per node, cycling by node in a round-robin fashion, ...). For a brute-force approach, 5760 benchmarks would be required to analyse all possible combinations of the 9 input parameters. The execution of Fesom2 was configured to use 32 cluster nodes with 16 physical cores each. In the past, Fesom2 was solely tuned manually. The best result from the first Fesom2 experiment showed that it is as good as results which were previously achieved based on expert knowledge and profiling, e.g.,

to manually tune MPI settings for the communication patterns detected.

For the second Fesom2 experiment, 10 MPI runtime parameters were tuned in addition to the 9 parameters of the first experiment (e.g., for the algorithm selection for collective MPI routines, handling of barriers, ...). With these additional MPI parameters, a brute-force approach would require several billion benchmarks to analyse all possible combinations of all input parameters. The execution of Fesom2 was configured again to use 32 cluster nodes, each with 16 physical cores. The best result from the second Fesom2 experiment is, in turn, comparable to the best result from the first experiment. This shows that the Black Box Optimizer automatically finds also equivalent settings for the MPI runtime environment that are comparable to the ones found by time-consuming manual tuning.

For detailed benchmark results refer to Deliverable 5.1 [Him19].

5.3.4 Applied Performance and Software Engineering Concepts

Listed below are the performance and software engineering concepts which were applied within the scope of this success story. For more information about these concepts refer to [HHKS18].

PE3 Benchmarking

Notes: controlled experiments were performed automatically to measure speedups and efficiencies by providing varying HPC resources, e.g., 1, 2, 4, 8, ... cores on a shared memory system or 1, 2, 4, 8, ... nodes on a distributed system

PE4.1 Tuning without Building a Parallel Program

Notes: BQCD was tuned via partitioning, tiling, and setting of internal runtime parameters

PE4.2 Tuning without Modifying the Source Code

Notes: both MPI test programs (π calculation and SAT solver) as well as Fesom2 were tuned by searching e.g., the best performing compiler and MPI environment, appropriate compiler/linker options, and appropriate runtime parameters

PE5 Optimization Cycle

Notes: in order to find the best parameter combination for building and/or running a parallel program the cycle *set next parameter combination* $\rightarrow$ *benchmark* is handled automatically by the Black Box Optimizer during the examination of various parameter combinations based on genetic algorithms

5.3.5 Material

The small MPI test program to calculate π is contained in the download package of the MPICH implementation of MPI [MPI19b]). For the small MPI test program to solve boolean satisfiability problems (SAT) see [Bur19] with reference to [Qui03]. The development of the BQCD (Berlin Quantum Chromodynamics) program was started in 1998 by Stüben for the two flavour case and the original Wilson action [ABS18]. The BQCD sources are available for download [ABS18]. For Fesom2 (Finite-Element/volumE Sea ice-Ocean Model), a multi-resolution ocean general circulation model that solves the equations of motion describing the ocean and sea ice using finite-element and finite-volume methods on unstructured computational grids see [Fes19]. The Fesom2 sources are available for download via GitHub [Fes19]. The Black Box Optimizer Tool goes back

to ideas used to find best input parameter values automatically with regard to a certain optimal behavior of a coupled system simulating the different processes at an airport [HKMW14]. The corresponding framework approach for the simulation based optimization using genetic algorithms was proposed in the context of the BMBF (Bundesministerium für Bildung und Forschung/Federal Ministry of Education and Research) cluster of excellence project “Efficient Airport 2030” [HKMW14].

Acknowledgement

The PeCoH project has received funding from the German Research Foundation (DFG) under grants LU 1353/12-1, OL 241/2-1, and RI 1068/7-1.

Bibliography

[ABS08] M. Allalen, M. Brehm, and H. Stüben. Performance of quantum chromodynamics (qcd) simulations on the SGI Altix. *Computational Methods in Science and Technology*, 14(2):69–75, 2008.

[ABS18] M. Allalen, M. Brehm, and H. Stüben. Berlin quantum chromodynamics program (BQCD) — source code download. `https://www.rrz.uni-hamburg.de/services/hpc/bqcd`, 2018.

[Bur19] John Burkardt. Circuit satisfiability using MPI — home page. `https://people.sc.fsu.edu/~jburkardt/c_src/satisfy_mpi/satisfy_mpi.html`, 2019.

[Ecl19] Eclipse. The platform for open innovation and collaboration – home page. `https://www.eclipse.org/„` 2019.

[Fes19] Fesom2. Finite-element/volume sea ice-ocean model — home page. `https://fesom.de/models/fesom20/`, 2019.

[GNU19] GNU. GNU Emacs an extensible, customizable, free/libre text editor – home page. `https://www.gnu.org/software/emacs/`, 2019.

[HHKS18] Kai Himstedt, Nathanael Hübbe, Julian Kunkel, and Hinnerk Stüben. An HPC certification program proposal meeting HPC users' varied backgrounds. Concept paper `https://wr.informatik.uni-hamburg.de/research/projects/pecoh/start`, DKRZ and RRZ, 2018.

[Him19] Kai Himstedt. D5.1 documentation of recommendations. Deliverable — online available via the PeCoH project webpage `https://wr.informatik.uni-hamburg.de/research/projects/pecoh/start`, 2019.

[HKMW14] Kai Himstedt, Steven Köhler, Dietmar P. F. Möller, and Jochen Wittmann. Ein Framework-Ansatz für die simulationsbasierte Optimierung auf High-Performance-Computing-Plattformen. In Jochen Wittmann and Dimitris K. Maretis, editors, *Simulation in Umwelt- und Geowissenschaften. Workshop Osnabrück 2014*, pages 109–122. Shaker Verlag, Aachen, 2014.

[Mic18] Microsoft. Visual Studio Code: Code editing. redefined – home page. `https://code.visualstudio.com`, 2018.

[MPI19a] MPI. Max-Planck-Institut für Meteorologie: CDI climate data interface – overview. `https://code.mpimet.mpg.de/projects/cdi`, 2019.

[MPI19b] MPICH. A high performance and widely portable implementation of the message passing interface (MPI) standard — home page. `https://www.mpich.org/`, 2019.

[Qui03] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group, 2003.

[SCH18] Martin Spindler, Victor Chernozhukov, and Christian Hansen. hdm: High-dimensional metrics. https://cran.r-project.org/web/packages/hdm/index.html, 2018.

[Sch19] Sandra Schröder. D2.2 code co-development. Deliverable — online available via the PeCoH project webpage https://wr.informatik.uni-hamburg.de/research/projects/pecoh/start, 2019.

[ubu19a] ubuntuusers. Nano wiki. https://wiki.ubuntuusers.de/Nano/, 2019.

[ubu19b] ubuntuusers. VIM wiki. https://wiki.ubuntuusers.de/VIM/, 2019.

[Uni19] Unidata. Data services and tools for geoscience — home page. https://www.unidata.ucar.edu/software/netcdf/, 2019.

[Urb18] Simon Urbanek. R Benchmark 2.5. http://r.research.att.com/benchmarks/R-benchmark-25.R, 2018.

[Wes17] Steve Weston. Introduction to doMPI. https://cran.r-project.org/web/packages/doMPI/vignettes/doMPI.pdf, 2017.