Universität Hamburg

DER FORSCHUNG | DER LEHRE | DER BILDUNG

D5.1 Documentation of Recommendations

Kai Himstedt and Hinnerk Stüben

Work Package:	WP5
Responsible Institution:	RRZ, DKRZ
Date of Submission:	November 2019

Abstract

The goal of the Performance Conscious HPC (PeCoH) project is to increase the runtime performance of parallel applications. The main focus is on tuning parallel programs 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. Tuning via reprogramming will be rather an exception in this work package (WP). It is described, how switching to an automatic tuning approach using a Black Box Optimizer tool (based on genetic algorithms) can greatly reduce the effort still needed in the beginning of the project for manually tuning parallel programs. Recommendations (lessons learned, best practices, ...) are given for the software packages we dealt with. Some observations from the benchmarks provide generally valid insights (e.g., the latest compiler generation is not always the fastest). On the basis of various benchmark results a short comparative assessment is given between the manual and the automatic tuning approach.

Contents

Chapter 1

Relation to the Project

The following text is an excerpt from the description of the project proposal for work package 5 (WP5):

> *High-level tuning possibilities of application software are evaluated and recommendations for efficient use are documented. This requires 1) the determination of tuning possibilities, 2) the set-up of realistic use cases for benchmarking, 3) benchmarking itself, and 4) documentation of the tuning solution. Application software, which is used RRZ and TUHH, shall be evaluated. [...] Note that we will not perform detailed performance optimization of the packages themselves, but focus on high-level configuration and runtime issues, for example, compiler flags, job placement and MPI settings.*

First of all, the significant aspects of the tasks of WP5 will be addressed: The help desk, which was set up as a ticketing system (see also [HK18], p. 11), is available for tuning questions and support (Task 5.1). The determination of tuning possibilities (Task 5.2) covers specific tuning options that can be set via environment variables, command line arguments, input keywords, parallelization (choice of message passing vs. threads, work decomposition, matching to compute hardware), and I/O (choice of file sizes and numbers of files, matching to I/O hardware). In order to study tuning possibilities in real benchmarks, realistic use cases have to be set up (Task 5.3). In principle, tuning settings can be guessed from experience. Real benchmarking (Task 5.4) is performed in addition in order to verify that a good tuning setting was found. For each software tuning possibilities, use cases, as well as benchmark results for different tuning settings are described, and recommendations are given (Task 5.5).

Chapter 2

Introduction

Users running well-tuned programs in a well-tuned environment will get their results faster, which certainly has a motivating effect. Additionally they will better exploit the HPC resources, which reduces the operating costs of the data center for the scientific outputs. This leads to a classic win-win situation.

We started our first tuning efforts approaching the users and prioritizing the activities to focus on the statistics package R. For two of the three studied use cases, we used a co-development approach to exploit the parallelization potential of sequential loops without data dependencies. A typical high level tuning was performed for the third use case, for which no source code was modified, by selecting particularly efficient libraries and by using a suitable combination of compiler and OpenMP/MPI environment. These use cases will be discussed in Chapter 3 on Traditional Tuning.

All further experiments are also based on tuning without or nearly without the source code being modified. This includes experiments to find good settings for using the standard software packages Gaussian [Gau19] and MATLAB [Mat19] in an HPC environment, which is also described in Chapter 3. For MATLAB experiments, we additionally parallelized sequential loops using the `parfor` paradigm. The tuning of runtime options from the outside without the need to (re-)build a parallel program can be considered as a sub-category of tuning without modifying the source code.

The typical tuning workflow is represented by an optimization cycle with the steps *set parameter combination*, *benchmark the program*, and *evalute benchmark result to determine the next improved parameter combination*. This approach was suitable to produce good tuning results, but nevertheless traditional manual tuning has turned out to be much more time consuming than anticipated. For this reason, we realigned the focus in WP5 to perform experiments with a Black Box Optimizer Tool in order to automatically find the parameter combination for a parallel application that gives the best benchmark result. This approach, which is based on genetic algorithms, is described in Chapter 4 on Automatic Tuning. 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.

Chapter 3

Traditional Tuning

3.1 Three Use Cases for R Programs

Use Case A

We initially started with manually tuning programs written in the language R, because the most urgent users' needs for tuning have arisen in this context. Three use cases for R programs were studied in order to give the specific tuning information relevant for the R software package. Tuning ideas were determined by expert knowledge and the software documentation and implemented subsequently.

Experiments were performed based on the R Benchmark 2.5 test suite [Urb18]. 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, ...).

The benchmark results are shown in Table 3.1 for experiments with two CPU types. It was observed that the initialization phase of a test (e.g., to setup large matrices) took quite some time, so the R Benchmark was additonally instrumented to show the time used for initialization and the actual computing time separately. A direct comparison of time results is only meaningful for experiments for which the same CPU types were used.

#	Machine	CPU (Intel)	Binary	Mathlib	Threads	Initialization Time [s]	Computing Time [s]	Total Time [s]
1	PC	Core i5-6500	package manager	Standard	2	71.03	120.17	191.20
2	PC	Core i5-6500	build from source	Standard	1	70.47	132.75	203.22
3	PC	Core i5-6500	build from source	Standard	2	69.07	119.89	188.96
4	PC	Core i5-6500	package manager	OpenBLAS	2	14.54	16.54	31.08
5	PC	Core i5-6500	build from source	OpenBLAS	1	15.83	19.20	35.03
6	PC	Core i5-6500	build from source	OpenBLAS	2	13.89	14.87	28.76
7	Cluster	Xeon E5-2630	package manager	OpenBLAS	1	19.46	19.45	38.91
8	Cluster	Xeon E5-2630	build from source	Standard	1	68.74	115.81	184.55
9	Cluster	Xeon E5-2630	build from source	OpenBLAS	1	21.53	22.22	43.75
10	Cluster	Xeon E5-2630	build from source	OpenBLAS	4	19.91	18.11	38.02
11	Cluster	Xeon E5-2630	build from source	OpenBLAS	16	19.98	17.38	37.36
12	Cluster	Xeon E5-2630	build from source	MKL	1	18.45	17.58	36.03
13	Cluster	Xeon E5-2630	build from source	MKL	4	16.61	14.22	30.83
14	Cluster	Xeon E5-2630	build from source	MKL	16	16.30	13.42	29.72

Table 3.1: Results for the R Benchmark 2.5 Test Suite

The results of the experiments show that the selection of an efficient library like OpenBLAS or MKL lead 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 were usually better than those achieved with OpenBLAS by a small margin. (For the benchmarks performed in the PC environment the MKL library was not available.) Using one core on a single cluster node, we achieved a speedup of about 5 for the configuration with the MKL library. Additional experiments were performed using the environment variable OMP_NUM_THREADS in order to exploit parallelism using several threads. But, using up to 16 cores on a single cluster node, the results in Table 3.1 show that hardly any additional speedup could be achieved (about 15% (OpenBLAS) and about 18% (MKL) compared to a single core). Increasing the number of cores reduced for both libraries the computing times slightly better than the initialization times. It seems that simple initialization instructions do not contain much exploitable parallelism.

Use Case B

Experiments were performed taking the rlassoEffects-function [SCH18] as a real-world example. For this use case, the parallelization was implemented by replacing sequential loops with parallel loops using appropriate R packages like doMPI, foreach, iterators, and Rmpi [Wes17]. The experiments have the general form of "$X \leftarrow matrix(rnorm(n \cdot p), ncol = p)$...". The benchmark results are shown in Table 3.2 for a small problem size and in Table 3.3 for a larger problem. To avoid measuring inaccuracies and to avoid effects caused by special features of current CPU architectures the values for $n, p, ...$ are chosen in a way that runtimes are not too short. 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, when for example 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 be usually reduced over time to avoid a rise in CPU temperature, which is heavily affected by the clock rates.

#	Nodes	Cores Used per Node	Total Cores Used	HT	CPU Time [s]	System Time [s]	Total Time [s]	Speedup	Efficiency
1	1	1	1	no	134.966	0.068	135.034	1.00	100.00%
2	1	2	2	no	66.485	0.068	66.553	2.03	101.45%
3	1	4	4	no	37.149	0.068	37.217	3.63	90.71%
4	1	8	8	no	21.532	0.059	21.591	6.25	78.18%
5	1	15	15	no	15.387	0.062	15.450	8.74	58.27%
6	1	31	31	yes	19.589	0.096	19.685	6.86	44.26%
7	2	16	31	no	14.283	0.150	14.432	9.36	30.18%
8	2	32	63	yes	18.534	0.315	18.849	7.16	22.74%
9	4	16	63	no	13.969	0.340	14.309	9.44	14.98%
10	4	32	127	yes	19.685	0.752	20.437	6.61	10.40%
11	8	16	127	no	15.279	0.710	15.989	8.45	6.65%
12	8	32	255	yes	19.819	1.880	21.699	6.22	4.88%

Table 3.2: Results for Benchmarking rlassoEffects (n=1000, p=400, s=20)

#	Nodes	Cores Used per Node	Total Cores Used	HT	CPU Time [s]	System Time [s]	Total Time [s]	Speedup	Efficiency
1	1	1	1	no	884.323	0.355	884.678	1.00	100.00%
2	1	2	2	no	438.495	0.316	438.811	2.02	100.80%
3	1	4	4	no	233.511	0.283	233.795	3.78	94.60%
4	1	8	8	no	124.181	0.293	124.474	7.11	88.84%
5	1	15	15	no	71.273	0.265	71.538	12.37	82.44%
6	1	31	31	yes	59.886	0.399	60.285	14.67	94.68%
7	2	16	31	no	37.275	0.340	37.615	23.52	75.87%
8	2	32	63	yes	44.339	0.564	44.904	19.70	62.54%
9	4	16	63	no	30.880	0.481	31.361	28.21	44.78%
10	4	32	127	yes	40.333	0.932	41.265	21.44	33.76%
11	8	16	127	no	30.048	0.814	30.862	28.67	22.57%
12	8	32	255	yes	34.167	2.074	36.240	24.41	19.15%

Table 3.3: Results for Benchmarking rlassoEffects (n=6000, p=600, s=20)

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 was executed via `mpirun`. For the sake of simplicity, this additional core is neglected in the tables for the calculation of speedups and efficiencies. In those experiments where the hyper-threading (HT) technology was enabled, the number of hyper-threaded cores is divided by two to give a fair calculation of the efficiency column, assuming that two hyper-threaded cores have only slightly more computing power than their corresponding physical core. Anyway, except for the case where only one node was used for the bigger problem size, the results show that hyper-threading has a negative impact on the speedups. In general the speedups and efficiencies 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, and 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. With 8 nodes hardly any improvement is achieved.

Use Case C

For the experiments, we parallelized – in collaboration with the user and base on the experience gained from Use Case B – a program for analyzing satellite night images using a foreach() paradigm. The performance engineering know-how was successfully transferred to end users. One new challenge was a large demand of the program for main memory. 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.

3.2 Using Gaussian in an HPC Environment

When using Gaussian [Gau19] it is worthwhile to care about setting up the execution environment. In particular, one can explixitly declare the number of CPU cores to be

used, how threads are pinned to these cores, the amount auf main memory that the program can use, and the directory for scratch files.

There are four different places where the environment can be specified (for an overview see [Gaua]): the *Default.Route* file, the input, the command line, or via environment variables. The latter three possibilities are considered in the following. Recommendations for concrete settings can be found at [Gaub].

CPUs (parallel execution) [Gauc]

By default Gaussian runs serially. For parallel execution the number of CPU cores to be used must be specified (cluster parallelism, i.e., using more than one node, is not covered here). Possibilities for CPU settings are shown in Table 3.4. Examples for CPU settings are given in Table 3.5.

input file	`%CPU=`*`list`*
command line	`-c=`*`list`*
environment variable	`GAUSS_CDEF=`*`list`*

Table 3.4: Possibilities for CPU settings in Gaussian.

list	`0,1,2,3,4,5`	
range	`0-5`	short for: `0,1,2,3,4,5`
range with increment	`0-5/2`	short for: `0,2,4`

Table 3.5: Examples for CPU settings in Gaussian (values of *`list`* in Table 3.4).

Specifying *`list`* has two effects: (a) The number of CPUs (or threads, respectively) is determined by the number of entries in *`list`*, (b) Gaussian tries to pin threads to the CPUs specified.

Main memory [Gaud]

By default Gaussian 2019 uses 800 MByte of main memory. To prevent swapping, it is important to specify more memory if needed. Possibilities for main memory settings are shown in Table 3.6.

input file	`%Mem=`*`value`*
command line	`-m=`*`value`*
environment variable	`GAUSS_MDEF=`*`value`*

Table 3.6: Possibilities for main memory settings in Gaussian.

giga-bytes	`50gb`	
mega-words	`100mw`	same as: `800mb`

Table 3.7: Examples for main memory settings in Gaussian (values of *`value`* in Table 3.6).

The amount of memory can be specified in multiples of `kb` (kilo-bytes), `mb` (mega-bytes), `gb` (giga-bytes), `tb` (tera-bytes), `kw` (kilo-words), `mw` (mega-words), `gw` (giga-words), `tw` (tera-words); 1 word = 8 bytes .

Directory for scratch files [Gaue]

The third performance influencing factor that should be mentioned is the location of scratch files. The directory for scratch files can be specified via the environment variable `GAUSS_SCRDIR`. The available space in that directory (or the corresponding disk) can be specified by a further environment variable: e.g., `GAUSS_RDEF="MaxDisk=95GB"`. The amount of space is specified in the same way as the amount of main memory (see above).

3.3 Using MATLAB in an HPC Environment

To speed up large computations (of any kind) parallelization is indispensable. In MATLAB [Mat19], parallel computing is enabled by the *Parallel Computing Toolbox*. It allows to use all compute resources of a single server. For employing multiple servers, the *MATLAB Parallel Server* module is needed, which will not be considered here.

MATLAB has a good introduction to parallel computing with their *Parallel Computing Toolbox*, see [Mata]. In general, parallelization in MATLAB is hard or impossible to obtain automatically. An exception is if most of the work of a program is done in a function that somebody has already parallelized. Then one instance of the existing program can "automatically" run in parallel. In many use cases, an existing program is run several times using different input and producing different output. Such runs are *independent* of each other and can, in principle, run at the same time in parallel. This is called *trivial parallelism*, *embarassingly parallel* or lately *pleasingly parallel*.

The idea of the *Parallel Computing Toolbox* is that only a single MATLAB license is needed for using the all cores of a compute node. A user could just start one instance of MATLAB on each core, but this would require one license per core. Therefore, MATLAB's special constructs for trivial parallelisation should be used. This requires to modify existing MATLAB programs: an outer loop has to be introduced that loops over different runs of the original program. The outer loop is a `parfor` loop (see [Matb], which also contains a simple example).

For running parallel MATLAB programs one needs to know how to set the number of workers (or cores) to be used. This requires the following steps:

- Open the MATLAB GUI and configure the `local` profile
  - select *Preferences*
  - select *Parallel Computing Toolbox*
  - enter the *Preferred number of workers in a parallel pool*
- The number of workers *n* can be set in the MATLAB script with the `parpool(`*n*`)` statement, see [Matc]. However, *n* cannot exceed the number set in the profile.

Chapter 4

Automatic Tuning Using a Black Box Optimizer Tool

In Section 3.1, it was shown in connection with the use cases for R programs, how the selection of an optimized Mathlib like OpenBLAS or MKL and an appropriate setting of runtime options can improve the performance of R programs. For these experiments, the various parameter combinations were manually examined. An obvious approach to avoid the hassle of this manual process would be to group the steps *set parameters* → *benchmark* (for experiments without the need to (re-)build the parallel program) or the steps *set parameters* → *build* → *benchmark* (if a build step is necessary) in order to automatically perform benchmarks for all parameter combinations in the sense of a brute-force approach. In principle, this could be done easily by the help of a small shell script, for example. For a small number of parameters, this is easily possible and is described, for example, in [GKJ17] for a simple climate time-stepped application based on stencil operations to update grid-bound variables. Parameters were varied for compiler selection (clang or gcc), the compiler optimization level and for the usage of Profile Guided Optimization (PGO) techniques. The emphasis there was on finding good parameter settings to reduce the compile times. But with an increasing number of parameters the search space increases exponentially and a brute-force search is then 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 are possibly application specific options, e.g., for partitioning and tiling. Due to this consideration and with the experience in manually tuning R programs we looked for a better solution and came up with the idea of using a Black Box Optimizer tool, which is based on genetic algorithms, to tune a wide range of optimization parameters automatically and efficiently. Such a tool was successfully 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].

In this case, the combination of the build- and subsequent benchmark step can be

regarded as a simulation model or black box, respectively. By choosing suitable input parameters, the Black Box Optimizer attempts to minimize the runtime of the parallel program. Such input parameters control the effective build step (compiler-, version- and MPI selection, optimization level, ...) and the further setting of suitable runtime options (e.g., to tune the MPI configuration). The build step is optional for pre-compiled applications and allows finding the best setting for application specific options, for example, for the domain decomposition by an appropriate partitioning and tiling.

For manual tuning, one typically iterates only over meaningful combinations of optimization parameters. But this would remain burdensome and requires above all expert knowledge. Promising parameter settings could be determined based on educated guesses or – if applicable – via time consuming profiling of the program. However, good combinations might get overlooked in manual analysis.

First, the basic principle of the simulation-based optimization method based on genetic algorithms shall be briefly explained.

4.1 Simulation-based Optimization with Genetic Algorithms

The ideas in this section go back to a framework approach which 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].

The simulation-based optimization has to find a set of values for the input parameters with regard to a given objective function. This function evaluates the simulation run in a certain way and delivers a measure for the quality of the current set of input parameter values. A very simple optimization problem might be to find optimal values for two integer parameters in the interval of 0 to 9. The first approach to solve the problem would be a brute-force algorithm that explores all possible combinations, runs the simulation model for all these (100) input vectors, calculates the objective function, and thus determines the optimum by a brute-force approach. But as already mentioned, such a brute-force approach is not suitable for practical use, because the number of combinations grows exponentially with the number of input parameters.

For real problems, more efficient search strategies are used, and genetic algorithms have proven successful in the past. Genetic algorithms mimic the process of natural evolution. They are part of the evolutionary algorithms and are used to find solutions to optimization problems by using techniques inspired by natural evolution, such as inheritance, mutation, selection, and crossover [Gol89]. In a genetic algorithm, a population of candidate solutions evolves towards better solutions. Mimicking the natural evolution, better solutions may be found without any knowledge of how to analytically solve the underlying problem. Zadeh [Zad94] introduced the term Soft Computing for methods such as fuzzy logic, artificial neural networks, and evolutionary algorithms to get a clear distinction from the classical Hard Computing, which is based on precision, certainty and analytical model. In contrast to the classical methods, Soft Computing is tolerant towards phenomena like vagueness, non-linearity or incomplete information, which are common for prediction problems.

Genetic algorithms are an appropriate method for solving 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 a still better solution by applying the genetic operations *crossover* (i.e., exchanging genes of the parents) and *mutation* (i.e., slightly

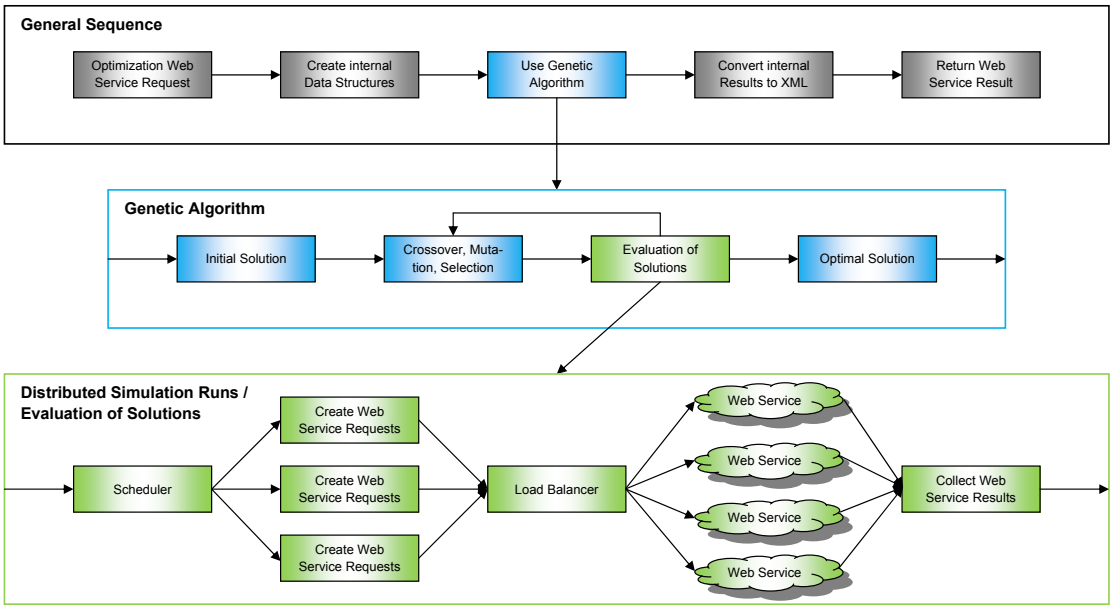

Figure 4.1: Framework Architecture of the Black Box Optimizer

modifying a gene with a low random probability). The optimization process starts with a set of (typically randomly created) individuals (known a *population*) which is named initial *solution*. New generations of solutions are produced until a sufficiently good solution has been found or the solution cannot be further improved. But nevertheless, the problem of exponential growth still remains for the more efficient search strategies. Furthermore, a single simulation run (e.g., building and benchmarking a parallel application for a specific parameter setting) usually requires a significant amount of CPU time. This leads to a demand for high performance computing systems and parallel algorithms for the simulation based optimization.

Figure 4.1 shows the framework architecture of the Black Box Optimizer. The optimization framework is based on Web Services [W319] and the use of XML standards for the exchange of structured data. Firstly, the input parameters of the optimization request are transformed to internal structures. The configuration for each simulation model is provided by the Web Service. The optimization step is performed by a genetic algorithm running identical simulation instances in parallel to determine the objective function for the different chromosomes. A scheduler creates the corresponding distributed Web Service requests and a load balancer component ensures a uniform utilization of the processing elements available in the cluster. The main benefit of the architecture lies in way that interfaces between these different levels are designed and especially in their independence from a specific simulation model and – in principle – the optimization algorithm used. The Black Box Optimizer is available as a Web Service itself in analogy to today's widespread "Everything as a service" paradigm [BFB+11].

4.2 First Tuning Experiments

The Black Box Optimizer tool is comfortable to use. The Web Services shown behind the load balancer in Figure 4.1 in Section 4.1 are implemented by a simple bash script that takes an actual setting of parameters on how to build and benchmark the parallel application to tune. After the simulation model has been made available as a Web Service,

it is simply required to configure the genetic algorithm (mutation probability, crossover probability, population size, generations, ...) and specify the Web Service interface description of the simulation model (i.e., to describe the parameters of the bash script). Furthermore, a list of worker components can be specified in order to execute identical copies of the bash script by the Black Box Optimizer in different working directories in parallel. A small number of worker components can be used to limit the number of simultaneously active jobs in the job queue in order to avoid using too many cluster nodes for the experiments at once. The workload manager used for the experiments was SLURM and jobs are submitted via a `sbatch` command, which is finally performed by the bash script to benchmark the parallel application.

With a first parallel test program to calculate π (contained in the download package of the MPICH implementation of MPI [MPI19]) a feasibility study was performed to examine the suitability of the Black Box Optimizer tool for an automated tuning of parallel applications in practice. Table 4.1 shows the input parameters we selected for this experiment and the meaning of their values.

#	Input Parameter	Value	Meaning
1	CompilerSelection	1	GCC
		2	Intel
		3	PGI
2	CompilerGeneration	1	latest, i.e. GCC 8, Intel 19, or PGI 18
		2	second latest, i.e. GCC 6, Intel 18, or PGI 17
		3	third latest, i.e. GCC 5, Intel 17, or PGI 16
		4	fourth latest, i.e. GCC 4, Intel 16, or PGI 15 (depending on CompilerSelection)
3	MPISelection	1	Open-MPI
		2	Intel MPI
4	OptimizationLevel	0	-O0
		1	-O1
		2	-O2
		3	-O3
		4	GCC: -Ofast, Intel: -fast, or PGI: -fast
5	UseProfileGuidedOptimzation	0	PGO off
		1	PGO on
6	UseHyperthreading	0	HT off
		1	HT on

Table 4.1: Input Parameters and Their Meaning for First Tuning Experiments

For a brute-force approach $480 = 3 \cdot 4 \cdot 2 \cdot 5 \cdot 2 \cdot 2$ benchmark runs would be required to analyse all possible combinations of the 6 input parameters. The execution of the parallel π program was configured to use four cluster nodes, each with 16 physical cores. For the genetic algorithm we chose a mutation probability of 5%, a crossover probability of 20%, and a population size of 20 as conservative defaults from the literature. Table 4.2 shows the best five chromosomes for generations 0..4 to give an impression of the search behavior of the algorithm.

Gen. #	Chrom. #	Compiler Selection	Compiler Generation	MPI Selection	Opt. Level	PGO	HT	Runtime Result [s]
0	1	GCC	third latest	Open-MPI	4	no	yes	157
	2	PGI	third latest	Open-MPI	0	yes	yes	176
	3	Intel	third latest	Intel MPI	1	no	yes	191
	4	Intel	fourth latest	Intel MPI	2	no	yes	194
	5	GCC	latest	Open-MPI	4	no	no	201
1	1	GCC	third latest	Open-MPI	4	no	yes	155
	2	PGI	second latest	Open-MPI	1	yes	yes	170
	3	PGI	second latest	Open-MPI	4	no	yes	181
	4	GCC	third latest	Intel MPI	4	no	yes	186
	5	Intel	third latest	Intel MPI	3	no	yes	187
2	1	GCC	third latest	Open-MPI	4	no	yes	154
	2	GCC	third latest	Open-MPI	4	no	yes	155
	3	GCC	third latest	Open-MPI	4	no	yes	156
	4	GCC	third latest	Open-MPI	4	no	yes	158
	5	GCC	third latest	Open-MPI	4	no	yes	160
3	1	GCC	second latest	Open-MPI	4	no	yes	151
	2	GCC	second latest	Open-MPI	4	no	yes	151
	3	GCC	second latest	Open-MPI	4	no	yes	152
	4	GCC	third latest	Open-MPI	4	no	yes	154
	5	GCC	third latest	Open-MPI	4	no	yes	154
4	1	GCC	second latest	Open-MPI	4	no	yes	149
	2	GCC	second latest	Open-MPI	4	no	yes	149
	3	GCC	third latest	Open-MPI	4	no	yes	152
	4	GCC	second latest	Open-MPI	4	no	yes	152
	5	GCC	second latest	Open-MPI	4	no	yes	152

Table 4.2: Best Five Chromosomes in Generations 0..4 for the Parallel π Calculation

The Black Box Optimizer already found a stable best solution in the 3rd generation, which is detected in Generation 4, after $100 = 5 \cdot 20$ benchmark runs for evaluating five populations in total. It should be mentioned, however, that it is in the nature of the genetic algorithm that it remains unknown if there exists a solution that is still better than the one found. In practice this is only of minor importance, because the optimal solution is usually only minimally better. For the tuning of the π application, we were not able to achieve a better result with a manual approach. The slight differences in the runtime results for individuals representing the same input parameters fall within typical differences when benchmarks are executed in a cluster system on different nodes of the same type as in our case.

While the π calculation was characterized by floating point calculations, we chose for the second test program a simple parallel solver for boolean satisfiability problems (SAT) from the artificial intelligence (AI) domain. It is based on integer arithmetic and does an exhaustive search of all 2^N possibilities for a logical function of N logical arguments (see [Bur19] with reference to [Qui03]). The execution of the solver was configured to use four cluster nodes, each with 16 physical cores. The configuration of the genetic algorithm (mutation probability 5%, crossover probability 20%, population size 20) and the 6 input parameters to optimize were the same as for the π calculation (resulting in 480 combinations, see also Table 4.1). The benchmark results are shown in Table 4.3.

Gen. #	Chrom. #	Compiler Selection	Compiler Generation	MPI Selection	Opt. Level	PGO	HT	Runtime Result [s
0	1	GCC	third latest	OpenMPI	4	no	yes	154
	2	Intel	third latest	Intel MPI	1	no	yes	192
	3	GCC	third latest	OpenMPI	3	yes	no	194
	4	PGI	third latest	OpenMPI	0	no	yes	196
	5	Intel	fourth latest	Intel MPI	2	no	yes	196
1	1	GCC	third latest	Intel MPI	1	yes	yes	151
	2	GCC	third latest	OpenMPI	4	no	yes	152
	3	GCC	third latest	Intel MPI	4	no	yes	153
	4	PGI	second latest	OpenMPI	4	no	yes	158
	5	Intel	third latest	Intel MPI	3	no	yes	184
2	1	GCC	third latest	Intel MPI	1	yes	yes	149
	2	GCC	third latest	Intel MPI	1	yes	yes	149
	3	GCC	third latest	Intel MPI	1	yes	yes	150
	4	GCC	third latest	OpenMPI	4	no	yes	151
	5	GCC	third latest	Intel MPI	3	no	yes	152

Table 4.3: Best Five Chromosomes in Generations 0..2 for the Parallel SAT Solver

This time the Black Box Optimizer found a stable best solution already in the 1st generation, which is detected in Generation 2 (by the confirmation of the best parameter setting in Generation 1), after a total of $60 = 3 \cdot 20$ benchmarks for evaluating three populations.

Because of the comparatively small search space in both experiments, good solutions were already found in Generation 0 within the first 20 randomly chosen parameter settings, so that only minor improvements were achievable for the succeding generations. Anyway, the comparison of the best chromosomes for the calculation of π and solving a SAT problem in Tables 4.2 and 4.3 shows by differences in the compiler generation, the MPI selection, the optimization level, and in the usage of PGO that genetic algorithms can be succesfully applied for tuning parallel programs from different domains.

4.3 Tuning Real Applications

The tuning results of the first two test programs were so promising that we decided to switch to the automatic tuning of real applications quite early.

4.3.1 BQCD

For our first use case, we selected the BQCD (Berlin Quantum Chromodynamics) program. BQCD is a Hybrid Monte Carlo program for simulating lattice QCD with dynamical Wilson fermions [HNS19]. The development of BQCD was started in 1998 by Stüben for the two flavour case and the original Wilson action [ABS18]. The sources are also available for download [ABS18]. BQCD can be regarded as a typical MPI program for which the parallelization is based on the domain decomposition of the underlying lattice. Of central importance for the performance is a suitable partitioning, i.e., the best mapping of work to processing elements. In contrast to the first two experiments, the build step is not included in the optimization. Table 4.1 shows the 8 input parameters we selected for the BQCD experiments and the ranges of their values. These parameters are handled internally by the BQCD program to control the mapping to processing elements and some other functionalities of the program. For the sake of simplicity, we shall not go into

more detail of their meaning. Indeed, it is not even necessary, because also the Black Box Optimizer modifies these parameters without any knowledge about their meaning.

#	Input Parameter	Value	Meaning
1	ProcessesPerNodeLog2	0..5	Giving ProcessesPerNode=$2^{\text{ProcessesPerNodeLog2}}$ for nodes supporting max. 32 Hyperthreads
2	P2DivisorIndex	0..5	Lattice Partitioning, Variable Dimension #1
3	P3DivisorIndex	0..5	Lattice Partitioning, Variable Dimension #2
4	HyperthreadMode	0..2	0=no, 1=one free, 2=use all
5	CGCloverMode	0..1	Internal BQCD Parameter
6	CGDMode	0..3	Internal BQCD Parameter
7	CGSpincolMode	0..1	Internal BQCD Parameter
8	UseSIMD	0..1	0=no, 1=yes

Table 4.4: Input Parameters and Their Value Ranges for the BQCD Tuning Experiments

For a brute-force approach $20736 = 6{\cdot}6{\cdot}6{\cdot}3{\cdot}2{\cdot}4{\cdot}2{\cdot}2$ 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. For the genetic algorithm we chose a mutation probability of 5%, a crossover probability of 20%, and a population size of 100 (again as conservative defaults from the literature). The number of generations was limited to 10. Table 4.5 shows the runtime results for the best, 10th best, 20th best, and 50th best chromosome for generations 0..10 to give an impression of the development of the runtime results.

Gen. #	Chrom. Pos. 1 (best)	Chrom. Pos. 10	Chrom. Pos. 20	Chrom. Pos. 50
0	55.38	76.93	95.50	<invalid>
1	56.37	69.30	76.71	113.00
2	56.43	65.23	68.48	81.66
3	52.17	59.22	63.58	71.19
4	50.11	56.57	59.54	67.11
5	50.08	52.17	57.05	61.44
6	50.26	51.14	56.11	60.17
7	49.63	49.88	50.48	55.46
8	49.48	49.89	50.26	51.19
9	49.62	50.18	50.40	51.31
10	49.99	50.45	50.66	51.11

Table 4.5: BQCD Runtime Results [s] for 4 Chromosome Positions in Generations 0..10

The best runtime result was achieved with $49.48s$ in Generation 8, which was about 10–15% faster than results achieved by manual parameter settings based on educated guesses by the author of the BQCD program. The input parameters for this result are shown in Table 4.6.

#	Input Parameter	Value	Meaning
	L1 (fixed)	32	Lattice Dimension 1
	L2 (fixed)	12	Lattice Dimension 2
	L3 (fixed)	12	Lattice Dimension 3
	L4 (fixed)	24	Lattice Dimension 4
1	ProcessesPerNodeLog2	4	Processes Per Node = 2^4 = 16
	P1DivisorIndex (fixed)	0	Index in [1, 2, 4, 8, 16, 32], i.e. P1 = 1
2	P2DivisorIndex	3	Index in [1, 2, 3, 4, 6, 12], i.e. P2 = 4
3	P3DivisorIndex	3	Index in [1, 2, 3, 4, 6, 12], i.e. P3 = 4
	P4 (calculated)	4	P4 = 4
4	HyperthreadMode	0	HT off
5	CGCloverMode	0	Index in [1, 2], i.e. Clover Value = 1
6	CGDMode	2	Index in [2, 21, 25, 35], i.e. D Value = 25
7	CGSpincolMode	1	Index in [1, 22], i.e. Spincol Value = 22
8	UseSIMD	1	SIMD Instructions Are Used

Table 4.6: Best Parameter Setting for the BQCD Tuning Experiments

The search space seems still to be comparatively small, so a solution almost as good was already found in Generation 4 and only minor improvements were achievable for the succeeding generations. Nonetheless, the results for the 10th best, 20th best, and 50th best results are coming closer together for the higher generation numbers. This indicates that there is at least a relative progress achieved, because the average fitness of the populations is still increasing. In general, this effect will be potentially positive for producing good descendants in further generations.

4.3.2 Fesom2

For the second use case, we selected the 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 [Fes19]. The sources are available for download via GitHub [Fes19]. 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. As with the optimization of the π and SAT programs a build step is involved to determine the best compiler, best compiler generation, and so on. Also considered for the tuning was the binding and mapping of processes to processing elements at runtime as well as the tuning of the MPI runtime environment to control for example how collective MPI operations are handled. In total, two Fesom2 experiments were carried out: For the first experiment the MPI parameters were tuned manually in advance and were additionally tuned by the Black Box Optimizer for the second experiment. Table 4.7 shows the 9 input parameters selected for the first Fesom2 experiment and the meaning of their values.

#	Input Parameter	Value	Meaning
1	CompilerSelection	1	GCC
		2	Intel
2	CompilerGeneration	1	latest, i.e. GCC 8 or Intel 19
		2	second latest, i.e. GCC 6 or Intel 18
		3	third latest, i.e. GCC 5 or Intel 17
			(depending on CompilerSelection)
3	MPISelection	1	Open-MPI
		2	Intel MPI
4	OptimizationLevel	2	-O2
		3	-O3
		4	GCC: -Ofast, Intel: -fast
5	UseProfileGuidedOptimzation	0	PGO off
		1	PGO on
6	UseHyperthreading	0	HT off
		1	HT on
7	MathlibSelection	1	MKL
		2	OpenBLAS
8	ProcessBindingSelection	0	Default (i.e. parameter will not be explicitly specified)
		1	none
		2	bind process to core
		3	bind process to hwthread
		4	bind process to socket
9	ProcessMappingSelection	0	Default (i.e. parameter will not be explicitly specified)
		1	blocked per node
		2	cycling by node in a round-robin fashion
		3	<ignored> (future extension)

Table 4.7: Input Parameters and Their Meaning for the First Fesom2 Experiment

For a brute-force approach $5760 = 2 \cdot 3 \cdot 2 \cdot 3 \cdot 2 \cdot 2 \cdot 2 \cdot 5 \cdot 4$ 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. For the genetic algorithm we choosed a mutation probabilty of 5%, a crossover probability of 20%, and a population size of 30 as conservative defaults. The number of generations was limited to 10. Table 4.8 shows the runtime results for the best chromosome for generations 0..10.

Gen. #	Chrom. Pos. 1 (best)
0	100.09
1	97.15
2	96.33
3	96.11
4	95.71
5	95.29
6	95.52
7	94.98
8	95.15
9	95.15
10	94.81

Table 4.8: Fesom2 Runtime Results [s] for Best Chromosome in Generations 0..10 (MPI Parameters still manually tuned)

The best runtime result was achieved with $94.81s$ in generation 10, which is a performance gain of about 5% in comparison to the best result in generation 0. The input

parameters for this result are shown in Table 4.9. The corresponding MPI parameters that were set manually are shown in Table 4.10.

#	Input Parameter	Value	Meaning
1	CompilerSelection	2	Intel
2	CompilerGeneration	2	second latest, i.e. Intel 18
3	MPISelection	2	Intel MPI
4	OptimizationLevel	3	-O3
5	UseProfileGuidedOptimzation	1	PGO on
6	UseHyperthreading	0	HT off
7	MathlibSelection	1	MKL
8	ProcessBindingSelection	0	Default (i.e. parameter will not be explicitly specified)
9	ProcessMappingSelection	0	Default (i.e. parameter will not be explicitly specified)

Table 4.9: Best Parameter Setting for the First Fesom2 Experiments

#	Input Parameter	Value	Meaning
1	I_MPI_FALLBACK	1	enable
2	I_MPI_LARGE_SCALE_THRESHOLDLog2	13	Threshold = 2^{13} = 8192
3	I_MPI_DYNAMIC_CONNECTION	1	enable

Table 4.10: Manually tuned MPI Parameters used for the First Fesom2 Experiments

The search space is still comparatively small, so a solution almost as good was already found in generation 5, and only minor improvements were achievable for the succeeding generations.

For the second Fesom2 experiment, several MPI runtime parameters were tuned in addition to the parameters shown in Table 4.7. These parameters and their value ranges are listed in Table 4.11 for Open-MPI and Table 4.12 for Intel MPI. For more detail, please refer to the Open-MPI or Intel MPI documentation.

#	Input Parameter	Value	Meaning
1	OMPI_MCA_coll	0..2	GCC
2	OMPI_MCA_coll_hcoll_enable	0..1	disable/enable
3	OMPI_MCA_coll_hcoll_priority	0..255	priority range
4	OMPI_MCA_coll_hcoll_npLog2	-1..5	-1 : Default (i.e. parameter will not be explicitly specified) 0..n: used for threshold calculation as 2^n
5	HCOLL_ENABLE_MCAST_ALL	0..1	disable/enable
6	HCOLL_ENABLE_MCAST	0..1	disable/enable
7	HCOLL_ML_DISABLE_BARRIER	0..1	disable/enable
8	HCOLL_ML_DISABLE_IBARRIER	0..1	disable/enable
9	HCOLL_ML_DISABLE_BCAST	0..1	disable/enable
10	HCOLL_ML_DISABLE_REDUCE	0..1	disable/enable

Table 4.11: Additional Tunable Open-MPI Parameters and Their Value Ranges for the Second Fesom2 Experiment

#	Input Parameter	Value	Meaning
1	I_MPI_FALLBACK	0..1	disable/enable
2	I_MPI_LARGE_SCALE_THRESHOLDLog2	-1..20	-1 : Default (i.e. parameter will not be explicitly specified) 0..n: used for threshold calculation as 2^n
3	I_MPI_DYNAMIC_CONNECTION	0..1	disable/enable
4	I_MPI_ADJUST_ALLGATHER	-1..5	-1 : Default (i.e. parameter will not be explicitly specified) 0: optimized default, 1..5: algorithm selection
5	I_MPI_ADJUST_ALLREDUCE	-1..12	-1 : Default (i.e. parameter will not be explicitly specified) 0: optimized default, 1..12: algorithm selection
6	I_MPI_ADJUST_BARRIER	-1..9	-1 : Default (i.e. parameter will not be explicitly specified) 0: optimized default, 1..9: algorithm selection
7	I_MPI_ADJUST_BCAST	-1..14	-1 : Default (i.e. parameter will not be explicitly specified) 0: optimized default, 1..14: algorithm selection
8	I_MPI_ADJUST_GATHER	-1..4	-1 : Default (i.e. parameter will not be explicitly specified) 0: optimized default, 1..4: algorithm selection
9	I_MPI_ADJUST_GATHERV	-1..3	-1 : Default (i.e. parameter will not be explicitly specified) 0: optimized default, 1..3: algorithm selection
10	I_MPI_ADJUST_REDUCE	-1..11	-1 : Default (i.e. parameter will not be explicitly specified) 0: optimized default, 1..11: algorithm selection

Table 4.12: Additional Tunable Intel MPI Parameters and Their Value Ranges for the Second Fesom2 Experiment

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. For the genetic algorithm we also chose, as before, a mutation probability of 5%, a crossover probability of 20%, and a population size of 150 as conservative defaults. The development of the runtimes was very similar to the results of the first experiment shown, in Table 4.8. In Generation 4 a runtime result of 94.84s was achieved (stated here as an individual result). The input parameters for this best result are shown in Table 4.13. The corresponding MPI parameters that were also tuned automatically are shown in Table 4.14.

#	Input Parameter	Value	Meaning
1	CompilerSelection	2	Intel
2	CompilerGeneration	2	second latest, i.e. Intel 18
3	MPISelection	2	Intel MPI
4	OptimizationLevel	3	-O3
5	UseProfileGuidedOptimzation	1	PGO on
6	UseHyperthreading	0	HT off
7	MathlibSelection	2	OpenBLAS
8	ProcessBindingSelection	0	Default (i.e. parameter will not be explicitly specified)
9	ProcessMappingSelection	0	Default (i.e. parameter will not be explicitly specified)

Table 4.13: Best Parameter Setting for the Second Fesom2 Experiments

#	Input Parameter	Value	Meaning
1	I_MPI_FALLBACK	1	enable
2	I_MPI_LARGE_SCALE_THRESHOLDLog2	15	Threshold = 2^{15} = 32768
3	I_MPI_DYNAMIC_CONNECTION	1	enable
4	I_MPI_ADJUST_ALLGATHER	-1	-1 : Default **(**i.e. parameter will not be explicitly specified**)**
5	I_MPI_ADJUST_ALLREDUCE	-1	-1 : Default **(**i.e. parameter will not be explicitly specified**)**
6	I_MPI_ADJUST_BARRIER	4	Algorithm: Topology aware recursive doubling
7	I_MPI_ADJUST_BCAST	5	Algorithm: Topology aware recursive doubling
8	I_MPI_ADJUST_GATHER	4	Algorithm: Binomial with segmentation
9	I_MPI_ADJUST_GATHERV	3	Algorithm: Knomial
10	I_MPI_ADJUST_REDUCE	-1	-1 : Default **(**i.e. parameter will not be explicitly specified**)**

Table 4.14: Best MPI Parameters setting for the Second Fesom2 Experiments

In the past, Fesom2 was solely tuned manually. The actual best result from the first Fesom2 experiment showed that the Black Box optimization yields comparable results

to those achieved on expert knowledge and profiling, e.g., to manually tune MPI settings for the communication patterns detected. 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 the time-consuming manual tuning.

Chapter 5

Summary and Conclusions

We have successfully started our study by manually improving the performance for several use cases based on the statistics package R, along with using efficient libraries like OpenBLAS or MKL, setting appropriate runtime options, and parallelizing loops with the foreach() paradigm in OpenMP and MPI environments. In Use Case A, the R Benchmark 2.5 test suite was employed. The a compiler optimization level of `-O3` gave the best results, whereas using Profile Guided Optimization (PGO) was not beneficial. The speedup results for MKL were usually minimally better than those achieved with OpenBLAS, whereas only little additional speedup could be achieved using several threads for the mathlibs (about 15% and 18% compared to a single core for OpenBLAS and MKL, respectively). Using one core on a single cluster node we achieved a speedup of nearly 5 compared to results achieved with the default mathlib used by the R package.

Experiments for Use Case B were carried out with the rlassoEffects-regression function and two problem sizes and for Use Case C with a program for analyzing satellite night images. For both use cases, the performance was improved by parallelizing appropriate loops with the foreach() paradigm. For the bigger problem size in Use Case B, we achieved a speedup of 30 on four cluster nodes, each node using 16 cores, and for Uses Case C a speedup of 126 on 32 cluster nodes, each node using four cores. A limiting factor was that R replicates its memory in every parallel instance. For Use Case B and C, we also conducted co-development with users to identify weaknesses in the performance of the existing implementation. The performance engineering know-how acquired this way was successfully transferred.

For Gaussian and MATLAB we have tested and documented how to run these programs in parallel.

For the examination of the three R use cases, the tuning workflows were performed manually. Although this was suitable to produce good tuning results, it turned out to be much more time consuming than anticipated. The reasons for this are obvious: the number of combinations grows up exponentially with the number of tuning parameters that can be modified and benchmarking all combinations of a huge search space is not possible in practice. Thus, typically only promising combinations are executed based on educated guesses and/or time consuming profiling the application. Furthermore, expert and domain specific knowledge is required for the manual approach and nevertheless good parameter combinations might still get overlooked. For this reason, we performed a series of experiments with a Black Box Optimizer tool, based on genetic algorithms, in order to tune parallel programs automatically. Genetic algorithms are widely used in practice as a strategy to reduce huge search spaces efficiently.

Initially, we successfully tuned a MPI test program to calculate π and another pro-

gram to solve boolean satisfiability problems. In comparison with the estimated effort of a brute-force search, the Black Box Optimizer found good solutions for the 6 tunable parameters examined (compiler selection, compiler generation selection, MPI selection, optimization level, PGO usage, and hyper-threading usage) almost immediately for both test programs. 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.

BQCD was selected as a first representative use case of a real MPI application that is based on an efficient domain decomposition of the underlying lattice and an appropriate mapping of work to processing elements of the cluster. For a brute-force approach, 20736 benchmark runs would have been required to analyse all possible combinations of the 8 parameters that were selected for tuning. The Black Box Optimizer found a good parameter setting already in Generation 8 after benchmarking 900 BQCD runs. The runtime of this parameter setting was actually about 10–15% shorter than runtimes achieved with parameter settings based on educated guesses.

Fesom2 was selected as a second representative use case of a real MPI application for which load imbalances caused by the lack of structure of the grids are typical. For the tuning the Black Box Optimizer was used again to find best parameters. This time, ten MPI runtime parameters were additionally tuned, which had previously to be done in a traditional way by the help of expert knowledge and profiling. This led to a resulting search space which contained several billion feasible solutions. Whilst in this case a brute-force search is out of the question, the Black Box Optimizer found in Generation 4 after executing 750 Fesom2 a parameter setting that are as good as the ones found in the past by manual tuning.

Originally, it was planned in WP5 to tune parallel programs also with regard to their I/O behavior, which did not apply to BQCD and Fesom2, because options for choice of file sizes and numbers of files, or use of I/O hardware only play a minor role for them. But as a result of the positive experience using a genetic algorithm to improve the performance of parallel programs so far it can be reasonably assumed that the performance of a parallel program providing appropriate I/O tuning parameters can be improved automatically in the same way.

To summarize the advantages of an automated tuning with the Black Box Optimizer in comparison with the traditional manual approach, it can be noted that genetic algorithms represent a generic approach to reduce huge search spaces drastically, that no expert knowledge for tuning is required anymore, and that the Black Box Optimizer was easy to use. Consideration must, however, be given to the fact that tuning with the new approach requires a certain amount of CPU resources for the benchmarks. Automated tuning will be particularly useful, if short-running benchmarks and a small number of cluster nodes are sufficient to tune the parallel program in a representative way, so the best parameter setting found also applies for long runtimes of the program using a larger number of cluster nodes. Break-even considerations can be used to assess the effort for the tuning in relation to the achievable savings expected.

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.


Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Bibliography

[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.

[BFB+11] P. Banerjee, R. Friedrich, C. Bash, P. Goldsack, B. Huberman, J. Manley, C. Patel, P. Ranganathan, and A. Veitch. Everything as a service: Powering the new information economy. *IEEE Computer*, 44(3):36–43, 2011.

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

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

[Gaua] Gaussian documentation. http://gaussian.com/options/?tabid=1.

[Gaub] Gaussian documentation. http://gaussian.com/relnotes/?tabid=3.

[Gauc] Gaussian documentation. http://gaussian.com/running/?tabid=4.

[Gaud] Gaussian documentation. http://gaussian.com/running/?tabid=3.

[Gaue] Gaussian documentation. http://gaussian.com/running/?tabid=0.

[Gau19] Gaussian. Expanding the limits of computational chemistry — home page. https://gaussian.com/, 2019.

[GKJ17] Anja Gerbes, Julian Kunkel, and Nabeeh Jumah. Intelligent Selection of Compiler Options to Optimize Compile Time and Performance — project poster. In *Euro LLVM 2017 (March 27 — 28, 2017). Saarbrücken, Germany*, 2017.

[Gol89] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[HK18] Kai Himstedt and Julian Kunkel. D1.1 annual report. Deliverable — online available via the PeCoH project webpage https://wr.informatik.uni-hamburg.de/research/projects/pecoh/start, 2018.

[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.

[HNS19] T.R. Haar, Y. Nakamura, and H. Stüben. BQCD manual. https://www.rrz.uni-hamburg.de/services/hpc/bqcd, 2019.

[Mata] Matlab documentation: Parallel computing. https://mathworks.com/help/parallel-computing/.

[Matb] Matlab documentation: parfor. https://mathworks.com/help/parallel-computing/parfor.html.

[Matc] Matlab documentation: parpool. https://mathworks.com/help/parallel-computing/parpool.html.

[Mat19] MatLab. Math. graphics. programming. — home page. https://www.mathworks.com/products/matlab, 2019.

[MPI19] 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.

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

[W319] W3. World Wide Web consortium – web services activity. https://www.w3.org/2002/ws, 2019.

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

[Zad94] Lotfi A. Zadeh. Fuzzy logic, neural networks, and soft computing. *Commun. ACM*, 37(3):77–84, March 1994.