European Community Seventh Framework Programme
Theme FP7-ICT-2011-7
Computing Systems

Automatic Online Tuning (AutoTune)

D5.1 The Selected Applications including the Achieved Improvements by Manual Tuning and the Tuning Effort

Renato Miceli (ICHEC)

Date of preparation (latest version): October 14th, 2012

PROJECT INFORMATION

Project acronym	AutoTune
Project full title	Automatic Online Tuning
Grant agreement no	288038
Call (part) identifier	FP7-ICT-2011-7
Funding scheme	Collaborative project

DOCUMENT INFORMATION

Deliverable Number	D5.1
Deliverable Name	The Selected Applications including the Achieved Improvements by Manual Tuning and the Tuning Effort
Authors	Renato Miceli (ICHEC), Carla Guillen (LRZ), Carmen Navarrete (LRZ), Antonio Pimenta (UAB), Anna Sikora (UAB), Laurent Morin (CAPS), Houssam Haitof (TUM), Enes Bajrovic (UNIVIE), François Bodin (CAPS), Martin Sandrieser (UNIVIE), Siegfried Benkner (UNIVIE)
Responsible Author	Mr. Renato Miceli Costa Ribeiro Irish Centre for High-End Computing 7th Floor, The Tower, Trinity Technology & Enterprise Campus Grand Canal Quay, Dublin 2, Ireland Email: renato.miceli@ichec.ie Phone: +353 1 5241608 ext. 41
Keywords	Periscope Tuning Framework; Application Repository; Evaluation
WP/Task	WP5 / Task 5.1

DISSEMINATION LEVEL

PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

COPYRIGHT NOTICES

ABSTRACT

This document D5.1, “Report on the selected applications including the achieved improvements by manual tuning and the tuning effort”, describes the work performed during months 01-12 of the AutoTune project, under task 5.1 of work package 5, “Application and Evaluation”.

The report presents the rationale behind and the approaches defined to steer and evaluate AutoTune, along with their respective challenges and limitations. Our focus herein lies on the application repository – the central workspace of test cases – and on the manual tuning of applications – the proof of concept for the tuning techniques. We conclude by discussing the tasks planned for the next development cycles of the project.

TABLE OF CONTENTS

1 Introduction

The goal of the AutoTune project is to simplify the development of efficient parallel programs on a wide range of architectures. For this, AutoTune will develop the Periscope Tuning Framework (PTF): an extension to Periscope – an existing application performance analysis tool – with plugins for performance and energy efficiency tuning. The resulting framework will be able to tune serial and parallel codes for multi- and many-core architectures and return tuning recommendations that can be integrated into the production version of the code. This whole tuning process will be performed automatically, i.e. each tuning plugin executes a set of techniques to optimize multi- and many-core applications.

While many tuning techniques exist, the technologies, libraries, patterns and architectures tackled by AutoTune have not been well explored in the context of automatic tuning, despite their relevance and extensive use in High Performance Computing (HPC). New techniques must therefore be developed and validated before their integration into the PTF framework. The possible interactions between the several techniques as well as the final framework require testing before becoming production-ready.

This document gives a report on the assessment and evaluation tasks of work package 5, "Applications and Evaluation", of the AutoTune project, as they stand in project month 12. The aim of this work package is to steer and evaluate AutoTune based on real workloads. In order to accomplish this, the following specific objectives shall be met:

- To set up an application repository;
- To analyze application bottlenecks and manually tune them;
- To evaluate the tuning components;
- To analyze common I/O bottlenecks for further AutoTune developments; and
- To develop a Best Practice user guide and documentation.

In this report, we present the rationale behind and the approaches defined to accomplish the aforementioned objectives, along with their respective challenges and limitations. Our focus herein lies on the tasks carried out under the project's development cycle 1 (months 01-12): the application repository, which is the central workspace of test cases; and the manual tuning of applications, which serves as proof of concept for the tuning techniques.

This report is part of Milestone M2 and covers the Project Phase 1 "Design and Demonstration" in its entirety (months 01-06); and partially Phase 2 "Development and Evaluation", whose end date is yet to be reached (months 07-32). The outcomes in this document were conducted under task 5.1 "Application repository and manual tuning", coordinated by the Irish Centre for High-End Computing (ICHEC).

This document targets a technical audience who intends to understand and contemplate the tasks relevant to the evaluation, practical analyses and assessment of AutoTune. We expect this document to clarify how we seek to use real-world applications to guide the development of effective tuning techniques, and to increase the user and developer confidence in the final framework for use in production runs.

The remainder of this document is organized as it follows. In Chapter 2 we describe the application repository, its intended use and structure, and the characteristics its applications should display. Chapter 3 presents the procedure we used to assemble the repository and details the application codes and data sets contained in the repository's first installment. Chapter 4 defines the approach taken by the tuning plugins to manually optimize the repository applications and the practical results achieved. Chapter 5 concludes the document with an analysis of the milestones accomplished during the first year of the project, while Chapter 6 discusses the tasks planned for the next cycles of the AutoTune project.

2 The Application Repository

The AutoTune project comprises the development of a set of tuning techniques and plugins for the PTF Framework. In order to aid in the process of validating the plugins, as well as proving the effectiveness and efficiency of the tuning techniques, we propose the Application Repository. We intend to use the Repository throughout the project's lifetime to guide the development and evaluation of the tuning plugins. As different use cases and scenarios are discovered, the Repository should be extended with the addition of new applications.

In this Chapter we discuss the Application Repository and its role within the project's evaluation and steering tasks. We approach the Repository agnostically to its contents and to the specific plugins they intend to assess; for a detailed analysis of the applications currently contained in the Repository, please refer to Chapter 3.

2.1 Description and Terminology

The Application Repository is a central workspace composed of representative and reliable inputs for test cases. Each input, herein named a Repository Application, consists of use cases of HPC software (mainly scientific simulations) and hardware (supercomputers, workstations and clusters). Consequently, an Application is a set of static resources that provides the system with stimuli, i.e. runs on the system.

A Repository Application is a tuple[1] of the following elements:

- Source code and configuration files
- Build instructions
- Input datasets
- Execution configurations

This composition uniquely exercises the Application's computational kernels. Computational kernels are the object of interest for tuning techniques and hence are the core input for our tuning plugins. Applications with multiple build instructions, input datasets and/or execution configurations stimulate their computational kernels differently and are therefore considered different inputs for test cases.

It must be noted that the Application Repository is merely a data structure; it does not establish how the data are employed to steer and assess the development of the plugins. It is a responsibility of the plugin developers to define the procedures to best adapt and exploit the Repository Applications, for every plugin has its own requirements and use guidelines which may not be general and could induce conflicts with the other plugins if the procedures are standardized.

In the remainder of this document, the Application Repository may be referred to as solely the "repository", whereas any Repository Application – i.e. an application contained within the repository – may be designated just as an "application".

2.2 Objectives

The repository is the toolbox used in the evaluation and assessment tasks within the project. Its content currently has a dual objective:

- To steer and guide the development of the tuning techniques and plugins; and
- To assess the quality of automatic tuning via the plugins compared to manual tuning.

The applications contained in the repository will provide a global view of the AutoTune behavior on scientific applications. They will also give hints about how the techniques and plugins should interact with the applications and tune them. As new use cases are discovered, they can be added to the repository in order to drive and enhance the development of the tuning techniques and plugins.

The feedback provided by manual and automatic tuning of repository applications is of utmost importance to the quality assessment of the techniques and plugins under development, during the design, implementation and integration phases. An initial quality assessment, described in Chapter 4, serves as a proof of concept

[1] In mathematics and computer science, a tuple is an ordered list of elements.

that the tuning techniques are worthwhile. Further manual and automatic tunings will aim to validate the effectiveness and efficiency of the tuning plugins compared to human developers.

Moreover, the repository will be used at the end of the project to evaluate the achievement of the project's goals. The improvements manually obtained, described in Chapter 4, will be compared to the improvements automatically achieved by the Periscope Tuning Framework. We expect that:

- PTF will obtain at least 50% of the manual improvements, and may even surpass them (>100%), due to its ability to more quickly explore the variant space; and
- PTF will require only a single or a few application runs, compared to effort timed in weeks or months for manual tuning.

The comparison of improvements between the manual and the automatic approaches will be undertaken as part of project task 5.3, by the end of the project (month 36).

In the future, the repository will also be used to assess strategies for multiple application aspects, such as for justified trade-off between energy and runtime tuning. Many applications display characteristics and performance issues that can be optimized both in terms of runtime and energy consumption; they can therefore be used as test cases for the decision and analysis of dependency between energy consumption and time performance. This specific work is planned for months 19-24 under project task 3.4, "Automatic energy efficiency analysis".

2.3 Features

In order to facilitate the achievement of the repository's objectives described in Section 2.2, we envisage the initial installment of repository applications display the features described in Table 1.

Feature	Rationale
Simple to build	The application must be able to run on at least one of the reference machines. Moreover, as each application will be used by multiple partners, a simple build process is necessary. Automatic build scripts are ideal.
Easy to execute	The application must be executable in a straightforward manner, to save the developers' time during the development and validation cycle. Ideally, the application can run both interactively (for plugin debug purposes) and in batch (for performance assessment purposes).
Execution time is short	A maximum wall time of 30 minutes is expected for the input data sets used to run the application. Otherwise, an iterative and constant validation process is not feasible, since an execution would take too long to provide the developers with feedback.
Licensing allows use, modification, sharing and redistribution	The applications contained in the repository must be available to all partners for use in the validation process, which includes execution and manual tuning. Redistribution access is desirable as part of the project's dissemination activity.
Result correctness can be easily checked	Regardless of the specific level of proficiency the developers have in an application and its domain, they must be able to evaluate whether the results achieved during the tuning procedure are valid. This implies having accessible input data sets and a known and simple procedure to check its conformance.
Performance is stable for a same execution scenario	Our auto-tuning techniques assume there is no significant difference in an application's performance for individual runs with the same resources and the same input data sets. Judging the best code variant becomes a complex

	decision if the application's performance varies over time.
Number of lines of code is moderate	A repository application ought to have at most 100,000 to 1,000,000 lines of code. Larger sums increase the probability that the application is hard to understand and to tune, both manually and automatically – since the tuning time also depends on the length of the source files. Smaller applications are more prone to be easier to learn (faster learning curve).
Test cases permit to steer and assess at least one plugin	The application's characteristics must be exploitable for guiding the development and validating at least one of the proposed tuning plugins.

Table 1: Expected features for repository applications

One should note that an application's moderate number of lines of code, or its simplicity to build an application and execute it for a short amount of time, do not render its use case plain or simplistic. The complexity of a use case depends on the steps and algorithms executed, not on the time each step and algorithm take. A short use case is equivalent to a long one as long as the same sequence of steps the application takes during its execution appears. Furthermore, the ability to quickly provide feedback to the developers is utterly important during the plugin design, implementation and integration phases. As the plugins and techniques become more mature, new input data sets can be added to cover for longer runtimes, should they not yet feature in the repository.

Here we presented the features we believe compose a good repository aimed at the project's evaluation and assessment. For a discussion of the application characteristics necessary for the specific needs of the plugins under development, refer to Section 3.1.

2.4 Structure

The repository is organized such that each application therein is independent from one another. In the repository's root directory lies a collection of subdirectories, one per application. An application is contained within its own subdirectory; different applications do not share any resources (input files, configuration files, permissions, etc.) and hard/symbolic links always point to a location inside that subdirectory. The objective of this approach is to avoid interferences and dependencies one application could cause in another one should they share resources. It is guaranteed that the set of files for an application is used exclusively for that application and no other one.

In an application directory one can find all the resources associated with that application, including but not limited to the source code, the input files, the configuration files, build and execution scripts (e.g. Makefiles and shell scripts). Every application directory has at least the following files:

- **LICENSE**, which defines the rights granted to the consortium for that application; and
- **README**, which describes the application, outlines its build and execution procedures and how to obtain metric measurements (execution runtime, energy consumption) for the application's computational kernels of interest.

2.5 Usage

The repository applications can be individually executed as independent test cases at any time during the development, validation, benchmarking and integration stages. These test cases are meant to uncover bugs and weaknesses the tuning techniques and plugins might display, both in terms of the features provided and of the tuning results achieved (time and/or energy consumption). When a test case succeeds for a specific plugin, it may be added to that plugin's regression testing suite.

A regression testing suite is a set of test cases that seeks to uncover regressions, i.e. new bugs, in software systems after a change is performed. In this case, the plugin's regression testing suite intends to ensure no defect reappears in the plugin's code while the tuning technique is under development. Thus, the regression testing suite comprises of only test cases that successfully execute for that plugin. If by chance a test case fails, the plugin developer is aware of the fact and can concentrate on the latest changes performed to the plugin since the suite was last executed. Therefore, constant execution of the regression test suite is a recommended good practice.

While regression testing suites are commonly used to uncover functional bugs inserted during development, they can also be used here to evaluate the progress of the tuning techniques as they evolve during the project's development cycle. It should be noted, however, that regressions in time or energy performance detected for specific applications do not necessarily mean that the tuning technique is receding. The technique and plugin developers are aware that changes performed to these products may have slightly negative impacts on the performance of certain applications, while rendering large positive overall gains to many other applications.

3 The Applications

In this Chapter we present the applications selected to compose the repository. We begin by characterizing our source population, as constrained by our auto-tuning framework and the plugins under development. We then introduce the process used to sample the population to compose our application repository; finally, we describe the applications selected for the repository, from a static perspective.

This Chapter describes the applications contained in the repository and the process used to obtain them agnostically to the repository itself and its use in the evaluation process. For a discussion focused on the application repository's merits, refer to Chapter 2.

3.1 Source Population

The population of all applications we can select for the repository is limited by the inputs the Periscope Tuning Framework accepts. PTF is based upon Periscope, a performance analysis tool developed by the Technische Universität München. Among its constraints, the following ones relate to the profiler's inputs:

- The source code must be written in C, C++ and/or Fortran;
- The form of computation can be serial or parallel;
- The application supports single- and multi-CPU;
- The execution can take place within a single or multiple nodes, using OpenMP and/or MPI.

The Periscope Tuning Framework is being built on top of Periscope, extended with tuning plugins. Therefore, PTF expands the input domain accepted by Periscope with the following characteristics, as defined by the plugins:

- Can be accelerated for GPGPUs using CUDA, OpenCL, HMPP and/or OpenACC directives;
- Can display parallel patterns such as pipelines, master-worker and data distribution.

The ability PTF has to work on GPGPU-accelerated applications also enlarged the set of supported architectures. The applications can now run on workstations and clusters with GPGPUs, such as CAPS' Nova and ICHEC's Stoney.[2]

Among the machines without GPGPUs, SuperMUC is the most prominent one, for being the 4th fastest supercomputer in the world[3] and a prospective extension is under consideration to include Intel Xeon Phi accelerators, which can execute OpenACC codes via CAPS HMPP compiler. Its speed and future inclusion of accelerators, along with its high availability and the dedicated support from LRZ make it the central

[2] For a detailed specification of the project's reference machines, refer to Appendix A: Specification of the reference machines, enclosed in this document.

[3] According to the Top500.org ranking, as of June 2012.

platform for the project development. Its migration machine, SuperMIG, served as a platform for future SuperMUC applications before this system's inauguration in July 2012.

Taking the aforementioned points into consideration, our source population is therefore composed of applications that feature any combination of the characteristics presented in Table 2.

Application characteristic	Options
Programming language	• C • C++ • Fortran
Parallel technology	• OpenMP • MPI • OpenCL • HMPP • CUDA • OpenACC
Parallel pattern	• Pipeline • Master-worker • Data distribution
Target system	• Machine with GPGPUs • Machine without GPGPUs • SuperMUC
Type	• Real application • Mini-application • Benchmark

Table 2: Characteristics of applications from our source population

The application type is also an important characteristic, given the topicality of auto-tuning approaches. Different application types aim at evaluating different uses for auto-tuning procedures:

- Real applications: tuned for better performance on an architecture where it is extensively executed;
- Mini-applications: a synthesized application composed of only a functional computational kernel, which demonstrates the tuning possibilities for applications that contain similar kernels (commonly from the same area of science); and
- Benchmarks: the widespread use of benchmarks to assess the quality of architectures suggests that the application will remain up-to-date and evolve to support future architectures; moreover, benchmarks provide many performance figures from different architectures, which can be used in comparative studies of auto-tuning performance.

These are the characteristics our source population has in order to work with the plugins proposed. For a discussion of the application features we desire for a good evaluation and assessment process, refer to Section 2.3.

3.2 The Sampling Process

Ideally, all applications that display the desirable features would be used to test and validate the tuning techniques and plugins. In practice, it is not feasible to experiment with every single application possible for every combination of features. Hence, we had to sample the population and extract a subset of applications for our validation purposes.

We guided our sampling procedure by strata. Each stratum is defined by one tuning technique, since at least one application for evaluating the technique and its plugin was needed.[4] Thus, our stratified sampling method defined the strata presented in Table 3.

Strata	Required Characteristics
High-level parallel patterns for GPGPU	• CUDA or OpenCL • Pipeline • Machine with GPGPUs
Hybrid manycore HMPP codelets	• HMPP or OpenACC • Data distribution • Machine with GPGPUs
Energy consumption via CPU frequency	• SuperMUC
Master-worker MPI	• MPI • Master-worker
MPI runtime	• MPI
Compiler flag selection	N/A

Table 3: Stratified application characteristics needed per tuning plugin

The compiler flag selection tuning is the only stratum that does not require any specific characteristic. This is due to the fact that every compiled application can compose a test case for this tuning technique and plugin. Provided that every C, C++ and Fortran application is compiled, no further characteristics must be enforced over an application from the source population.

The stratum for energy consumption tuning is one special case. These applications need to be run on the SuperMUC machine, since the framework for energy measurements is being developed and supported by IBM for the SuperMUC machine's Intel processors based on the Sandy Bridge architecture. Although the tuning technique can be generically applied, the plugin utilizes this framework for energy measurements; thus, its prototype, developed in collaboration with IBM, is intended for execution on SuperMUC, making this machine an initial requirement for applications exploitable by such plugin.

We must note that we were not looking for applications that fall into as many strata as possible. We acknowledge that the more tuning plugins are covered, the more complex the application is. Thus, its learning curve for building, executing and tuning shows a slower progress, yielding a longer time-to-solution. Besides, different parallel paradigms within the same application might interfere with each other and bias or constrain the tuning technique; for example, the core tuning problems may reside in a code region dominated by a paradigm the tuning plugin cannot deal with. For these reasons, we decided to ensure that every application is added to the repository with one tuning plugin assigned to it. Nonetheless, having

[4] For a better understanding of the requirements each tuning technique poses, refer to Document D4.1, "Design of the Tuning Plugins".

applications exploitable by several plugins is very beneficial since the interaction among tuning plugins is a study to be conducted in the future.[5]

Ultimately, the initial set of applications we selected for each of the aforementioned strata is composed of applications we are familiar with. As every application is meant to be experimented especially on a particular plugin, this plugin's developers are the main ones concerned about the application's ease to build and use and its facility to gather and validate results[6]. With that said, we decided to choose renowned and new applications we are familiar with and know they address these concerns. Once they are bypassed for one specific group of plugin developers the concerns cease to be risks – the internal intensive collaboration between partners contribute to spread knowledge specific to a certain application, and also gather feedback about the usefulness of the application for other plugins. That being said, one should be cognizant that the current repository is not final and more applications are planned for the future.

We must also mention that the applications chosen are not meant to be representative of their respective characteristic, algorithm or area of science. They were chosen merely due to their usefulness within our evaluation and assessment procedures. A thorough study of representative applications per characteristic, algorithm and area of science would yield an enormous application repository, impractical for the purposes and objectives we outlined in a timely manner.

3.3 The Selected Applications

In this Section we describe the applications we selected for inclusion into our repository. The repository currently contains 9 applications, from a multitude of domains and areas of science. Each subsection presents one application in the repository, along with its features:

- Brief description of the application, along with its purpose, domain and supported platforms;
- Characterization of the application's computational kernels of interest (e.g. algorithms, structure, parallel paradigms);
- Rationale about why the application was chosen and what tuning plugin can mainly benefit from it; and
- Representation of the input data sets (e.g. files, execution flags) and depiction of the resulting explored code.

A summary of each application's features comes at the end of each subsection.

This Section analyzes the selected applications from a static point of view, i.e. without executing them. We study the applications' features and behavior solely by examining each application's resources (e.g. source code, input data sets), thus no performance figure is gathered. For a study of the applications from a dynamic point of view – i.e. based on elements analyzed during runtime – and for performance figures (time or energy consumption), refer to Chapter 4.

3.3.1 BLAST

BLAST is a bioinformatics tool used to compare biological sequences (nucleotides and proteins) against databases of known ones. It searches for regions of similarity in biological queries and calculates the statistical significance of each match, comparing the input query with large databases of sequences such as GenBank or Swiss Prot. Based on heuristics, the BLAST algorithm improves up to 10 times the renowned exact-match Smith-Waterman algorithm.

BLAST is both a CPU- and a data-intensive application that can be executed in parallel and distributed systems, due to its embarrassingly parallel workload. When multiple processes are used, BLAST employs the Master-Worker paradigm, whereby the workload is divided among a pool of independent processes. The

[5] For the future works plans during the development and analysis of tuning plugins, refer to Document D4.1, "Design of the Tuning Plugins".

[6] Desirable features are defined in Section 2.3.

main algorithms for masking and filtering the sequences are very complex and thus provide a good subject for automatic tuning.

The Master-Worker structure within BLAST was developed by Claudia Rosas[7] from the CAOS department at UAB. It consists of a C++ mini-application that launches individual instances of BLAST to search across partitions of the main database.

The fact that the database can be arbitrarily partitioned (e.g. in chunks of same size) and the variability in computational complexity between partitions – rendering different execution times for each worker – makes BLAST an ideal subject for the Master-Worker model. The difference in execution time is caused by the variable similarity between the query and the chunk of database, being more time-consuming as there are more alignments. The tuning technique can explore the different chunk sizes evaluating how the application behaves, and may redistribute the load as desired by merging or splitting chunks.

BLAST has 2 main inputs: the sequence to compare, and the database containing a large group of known sequences to compare the query against. The sequence databases are normally dozens of GiB in size, while the queries are smaller files of several MiB. For our tests, we used the *nt* (nucleotide) database, available from the *ncbi* FTP server, and partitioned it into different configurations (replicated for each partition factor). The database used must be already partitioned and each of the fragments formatted independently.

For the query, we selected a random sequence of 1,036,416 characters extracted from the same database, which we call Input A. The large quantity of characters compared to the *nt* database results in a high alignment rate with the database contents and hence renders the query computationally-intensive, which is an appropriate scenario for the tests we will undertake.

Short Name	**BLAST**
Full Name	Master Worker Basic Local Alignment Search Tool
Purpose	Compare biological sequences
Area of Science	Biology, Bioinformatics
Main Algorithms	BLAST, DUST, SEG and XNU
Total Lines of Code	111,841
Platforms	Linux systems
Language	C++
Parallel Paradigm	Master-Worker (MPI)
External Package Dependencies	MPI (for the master worker framework)
Mainly Used By	Academia, Research
Related Works	N/A
Suggested By	UAB
Partner's Level of Expertise	Basic
Developed By	Stephen Altschul, Warren Gish, Webb Miller, Eugene Myers, and David J. Lipman (NIH – National Institutes of Health)
Main Referenced Publication	"Basic local alignment search tool". S. Altschul, W. Gish, W. Miller, E. Myers, D. Lipman. Journal of Molecular Biology 215 (3), pp 403-410, October 1990. doi: 10.1016/S0022-2836(05)80360-2

[7] *"Performance Improvement Methodology based on Divisible Load Theory for Data Intensive Applications", C. Rosas. PhD thesis, Universitat Autònoma de Barcelona, 2012*

Version	2.2.22
License	Public domain
Website	http://blast.ncbi.nlm.nih.gov/Blast.cgi

3.3.2 Convolution

Convolution is a toy application that performs transformations of images in the TIFF format using a simple convolution (a type of stencil computation). The application is targeted at image processing, although stencil computations are used on a wide set of scientific applications like electromagnetics, fluid dynamics and heat diffusion. Convolution is composed of about 1,000 lines of C code designed to run on Unix systems.

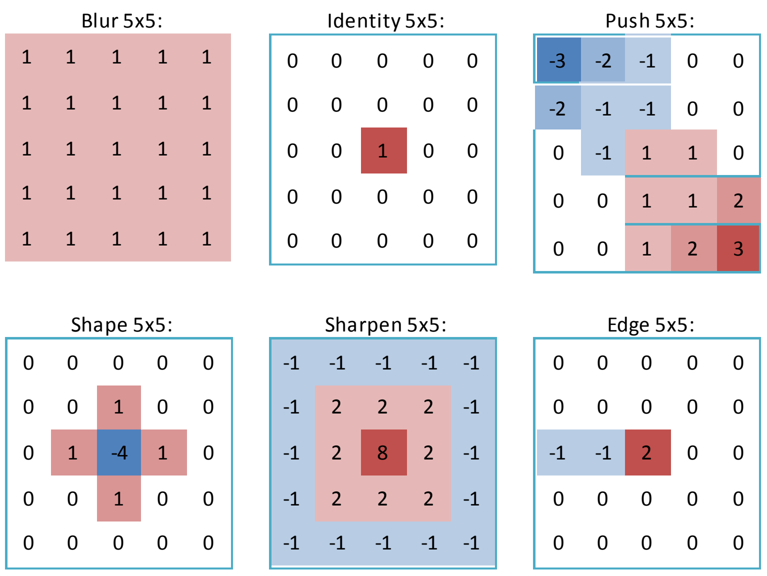
The application contains a single kernel: the convolution kernel. It performs an image transformation, which consists of a computation over a multi-dimensional space using neighboring points in the space and a field called “stencil”. The stencil is a square matrix characterized by a stride: the maximum offset of all neighbors. Mathematically, a convolution is defined as the integral of the product of a function with another function reversed and shifted; for the purposes of our study, a basic visual definition suffices.

A basic convolution has the asymptotic time complexity of *O(nd)*, where *n* is the grid size and *d* is the number of dimensions. The implementation of the naïve stencil computation takes around 25 lines and is made of a weighted sum of the values around the computed point. Equation 1 shows the mathematical definition of an image convolution, where *Pi,j* is the point in coordinates *(i, j)* in a grid, *F* is the convolution field; and *s* the stencil stride.

$$\sum_{i}\sum_{j}\sum_{k\in[-s,+s]}\sum_{l\in[-s,+s]} F_{k,l} \bullet P_{i+k,j+l}$$

Equation 1: Calculation of an image convolution

In Convolution, two image transformations are performed in a row using two different convolution fields. We implemented several 5-sized fields. The fields are chosen statically in the source code, among those in Figure 1.



Blur 5x5:

1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1

Identity 5x5:

0	0	0	0	0
0	0	0	0	0
0	0	1	0	0
0	0	0	0	0
0	0	0	0	0

Push 5x5:

-3	-2	-1	0	0
-2	-1	-1	0	0
0	-1	1	1	0
0	0	1	1	2
0	0	1	2	3

Shape 5x5:

0	0	0	0	0
0	0	1	0	0
0	1	-4	1	0
0	0	1	0	0
0	0	0	0	0

Sharpen 5x5:

-1	-1	-1	-1	-1
-1	2	2	2	-1
-1	2	8	2	-1
-1	2	2	2	-1
-1	-1	-1	-1	-1

Edge 5x5:

0	0	0	0	0
0	0	0	0	0
-1	-1	2	0	0
0	0	0	0	0
0	0	0	0	0

Figure 1: Several 5x5 convolution matrices

The application also supports fields of size 3; however, this support is disabled by default.

The result of each convolution is stored on a new grid or image, hence rendering the computation fully parallel: the input image and the stencil are read-only and the output image is write-only. When the application computes the first convolution, the algorithm's input image is the user's image provided as input to the application and the output is a scratch area used as an input for the second convolution. The user's image is then used as an output for this second convolution.

An expansion factor is available to arbitrarily increase the size of the input image and the amount of computation performed. At the beginning, the expansion consists of duplicating the input image over the constant factor on the *x* dimension. At the end of the computation, the image is shrunk to its original size by the mean of each point of the duplicated images.

By nature, a convolution stresses the memory hierarchy in a very particular way that has an impact on the performance: the geographic locality of the data read upon the computation of an image point is low (the memory being linear, the distance between the data read can be high) but is compensated by a good temporal locality (most data are read in consecutive iterations). The performance depends on the architectural specification, like the cache size, the register bank size, and features of the memory controller.

In the literature, several techniques have been introduced to improve the performance of convolutions for CPUs and GPUs. Among these techniques, loop tiling is efficient to adjust convolution computations on a cache hierarchy; loop unrolling increases the memory reuse; and adjustments to the grid size improve the memory controller bandwidths, especially on GPUs. These techniques are common for being parameterized by a variable: respectively, the tiling size, the unrolling factor, and the grid size. Therefore, they require a fine-tuning of these parameters for each machine, all of them depending on both the behavior of the compilation chain and the design of the architecture.

Convolution has all the properties required for the application repository: it is simple to use, simple to validate, and the performance is very sensible to the architecture used. It is furthermore highly representative of a family of kernels used in scientific applications. The application is provided with an OpenMP version and a HMPP version, making it ideal for the experimenting with the HMPP codelet tuning plugin.

Short Name	Convolution
Full Name	TIFF Convolution Filter
Purpose	Toy application that applies convolution filters to TIFF images
Area of Science	Image Processing
Main Algorithms	Convolution
Total Lines of Code	813
Platforms	Linux workstations and clusters, NVIDIA GPUs
Language	C
Parallel Paradigm	OpenMP, MPI, HMPP
External Package Dependencies	MPI, HMPP 3
Mainly Used By	Industry
Related Works	N/A
Suggested By	CAPS
Partner's Level of Expertise	Basic
Developed By	CAPS
Main Referenced Publication	N/A
Version	1.2
License	LGPL
Website	N/A

3.3.3 FaceDetect

We consider auto-tuning of pipeline patterns for heterogeneous manycore architectures. A pipeline is realized based on *while-loops* and corresponding source-code annotations. This annotated high-level code is then transformed by a source-to-source transformation tool into code that utilizes a heterogeneous runtime system for parallel execution on heterogeneous manycore architectures. The associated programming framework for high-level patterns (component framework, transformation system, coordination layer and runtime support) has been developed in the EU project PEPPHER[8].

Within the AutoTune project our goal is to develop auto-tuning techniques for optimizing such high-level pipeline applications for CPU/GPU-based systems. As a demonstrator application we will use an image-processing code that performs face detection on a stream of images in a pipelined manner, where the main image-processing components are taken from the OpenCV image-processing library.

In our framework, a pipeline consists of multiple stages where each stage usually corresponds to a function call. These stage functions are realized as multi-architectural components, i.e. for each function multiple implementation variants tailored for different execution units of heterogeneous target architecture (e.g. CPU and GPU variants) may exist. Our framework provides annotations that enable the user to specify the replication factor of individual stages as well as the buffer size for ports – which are then automatically generated by the transformation tool.

[8] *"PEPPHER: Efficient and Productive Usage of Hybrid Computing Systems". S. Benkner, S. Pllana, J. L. Traeff, P. Tsigas, U. Dolinsky, C. Augonnet, B. Bachmayer, C. Kessler, D. Moloney, V. Osipov, IEEE Micro, vol. 31, no. 5, pp. 28-41, Sep./Oct. 2011, doi:10.1109/MM.2011.67*

An example of a high-level pipeline pattern is given in Figure 2.

```
#pragma pph pipeline
while ( data.size != 0 ) {
     func1 ( iFile , data ); // connect func1 to func2 via data
     #pragma pph stage replicate (4) // replicate stage 4 times
     func2 ( data , cdata ); // connect func2 to func3 via cdata
     func3 ( cdata , oFile );
}
```

Figure 2: Example of a high-level pipeline pattern

A source-to-source transformation tool converts such a high-level representation into a representation that utilizes a pipeline coordination layer and a heterogeneous runtime system to coordinate and schedule the execution on concrete target architectures. The main tasks of the coordination layer and runtime system is to decide which implementation variants for stages should be executed on which execution units of a heterogeneous system.

We use the StarPU heterogeneous runtime-system[9] for the automatic scheduling of such computational tasks to available execution units. As there is a data dependency between pipeline stages, the utilization of resources depends not only on the parameters provided about the architecture (such as number of CPUs and GPUs) but also on the number of spawned tasks and replication factors of the pipeline stages as well as the scheduling policy used by the runtime system. Hence, realizing this application on heterogeneous multi-core systems provides various opportunities for automatic tuning. On the one hand, we expect performance limitations if the number of pipeline stages and the number of tasks is smaller than the number of available execution units; on the other hand, careful oversubscription could improve the application performance.

Within the AutoTune project we focus on auto-tuning support for structural aspects of a pipeline, most notably the replication factor of stages. Moreover, we will devise tuning support for certain aspects of the underlying coordination and/or runtime layer.

Figure 3 gives an impression of a possible high-level representation of FaceDetect, an image-processing pipeline for face detection based on OpenCV library routines.

```
unsigned int N = get_max_execution_units ();
#pragma pph pipeline with buffer ( PRIORITY ,N *2)
while ( inputstream >> file ) {
     readImage ( file , image );
     #pragma pph stage replicate (N) {
          resizeAndColorConvert ( image );
          detectFace ( image , outimage );
     }
     writeFaceDetectedImage ( file , outimage );
}
```

Figure 3: FaceDetect's pipelined high-level representation

[9] *"StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures". C. Augonnet, S. Thibault, R. Namyst, P.-A. Wacrenier. Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009, 23:187–198, 2011*

The stage functions `resizeAndColorConvert()` and `detectFace()` are realized as multi-architectural components with implementation variants for CPUs and GPUs, re-engineered from the OpenCV library.

The Open Source Computer Vision (OpenCV) is an open-source computer vision and machine-learning library with a strong focus on real-time applications[10]. Although the library is written in C and C++, it features interfaces for Python and Java and provides support for multicore processors as well as for CUDA-enabled GPUs. Moreover, OpenCV can take advantage of various libraries such as the Intel Integrated Performance Primitives (IPP) and Intel Thread Building Blocks (TBB).

Essentially, FaceDetect processes an input data stream, which usually consists of a series of video frames or a set of images of variable or constant resolution. The produced outputs are images with annotations (rectangles) signifying the position of the detected faces. We split the whole process into three stages: *read*, *process/detect*, and *write*. The *read* stage loads the image from the input data source (usually a file). The compute intensive part of the face detection is contained in the middle stage, where the application performs color conversion, optional resizing and face detection. However, the color conversion – which converts a colored image to grayscale – and the resizing are relatively fast compared to the face detection algorithm based on the Viola-Jones approach[11]. Since this detection algorithm relies on trained object models, the middle stage takes care of loading pre-trained object models as well. Finally, the *write* stage annotates images with rectangles and writes them to the disk.

FaceDetect's most notable test dataset is "Image dataset 1", which includes 400 images: 200 images in 1024x768 (XGA) resolution and 200 images in 2048x1536 (QXGA) resolution. The full paths to the image files can be specified in a textual input file; however, the data for this dataset ships with all 400 images contained in the "InputImages" folder. We also use the pre-trained model (classifier) "*haarcascade_frontalface_alt.xml*" that comes with the OpenCV library.

The different input data sets are known to affect FaceDetect's performance results: the higher the image resolution is, the more processing time is required. In addition, different pre-trained models can influence the performance as well as the detection rate. The tuning plugin for high-level parallel patterns for GPGPU can explore various combinations of parameters, including the number of CPUs and GPUs and parameters relative to the pipeline and to the scheduling policies. Essentially, we aim at maximizing the pipeline throughput, which translates to minimizing the overall execution time.

Short Name	**FaceDetect**
Full Name	Face Detection
Purpose	Demonstrator application for tuning of high-level pipeline patterns on single-node CPU/GPU-based architectures.
Area of Science	Computer Vision
Main Algorithms	Face detection based on Viola-Jones detector from the OpenCV library
Total Lines of Code	785
Platforms	Linux platforms with/without GPUs
Language	C++
Parallel Paradigm	Task, data and pipeline parallelism, TBB, pThreads, CUDA
External Package Dependencies	OpenCV, GCC 4.3+, TBB 2.2+, CUDA (if GPUs are available), pkg-config
Mainly Used By	Academia, Industry

[10] *"Learning OpenCV: Computer Vision with the OpenCV Library", G. Bradski, A. Kaehler. O'Reilly Media, 1st edition, October 2008*
[11] *"Rapid Object Detection Using a Boosted Cascade of Simple Features", P. Viola, M. J. Jones. IEEE*

Related Works	StreamIt and many other approaches for supporting pipeline parallelism (although most of these approaches do not address hybrid CPU/GPU systems)
Suggested By	UNIVIE
Partner's Level of Expertise	Intermediate
Developed By	OpenCV was initially developed by Intel. The high-level framework for pipeline patterns has been developed by UNIVIE, partly within the EU project PEPPHER.
Main Referenced Publication	"Rapid Object Detection Using a Boosted Cascade of Simple Features", P. Viola, M. J. Jones. IEEE CVPR (2001)
Version	1.0
License	GPLv3
Website	N/A

3.3.4 FSSIM

The Fish School Simulator (FSSIM) is a biological system simulator, which models the behavior of fish in an enclosed three-dimensional space. This application was developed by the CAOS Department at UAB[12] and has undergone previous optimizations to make it more scalable. The most important optimization consists in partitioning the workload in dynamic clusters of individuals that are close to each other instead of using spatial partition; this results in a more balanced execution and improves performance.

The simulator is individual-oriented, meaning that many instances of the subjects are created and set in the simulation space. Each individual moves and acts based on a few rules that determine its behavior. The main algorithm consists of a loop that computes the position of each fish based on the previous positions and the basic interactions between fish given by the modeled behavior. All processes are kept up-to-date about the position of nearby fish and a phase of information exchange is needed prior to computing the fish movement; both of those actions result in MPI communications. Moreover, migrations might occur among clusters of fish, which triggers inter-process communications.

FSSIM is an SPMD application that has intense communication phases and thus is a perfect test program for the MPI runtime tuning plugin. There are 3 different types of communication throughout the application: first of all, the distribution step, in which a small amount of large messages are exchanged; then, the information exchange between neighboring fish, which generates a large amount of small messages; and finally, the migration of fish, which generates medium-sized messages. The plugin will be able to tune different configuration parameters of MPI that yield changes in performance; for example, some parameters depend on the characterization of messages in the application. We expect the tuning will adapt to suit the predominant type of communication.

FSSIM is a relatively simple application that can run in small systems. This allows us to test many scenarios within a reasonable time period.

As input FSSIM requires an initial population of fish with their initial position, speed and orientation in the simulation space. The size of the input depends on the amount of fish the user wishes to simulate and ranges from a few thousand to half a million individuals.

[12] *"Particionamiento y Balance de Carga en Simulaciones Distribuidas de Bancos de Peces", R. Solar. PhD thesis, Universitat Autònoma de Barcelona, 2012*

Short Name	FSSIM
Full Name	Fish School Simulator
Purpose	Simulate the behavior of large groups of fish
Area of Science	Biology
Main Algorithms	Huth and Wissel model
Total Lines of Code	1,670
Platforms	Linux clusters
Language	C++
Parallel Paradigm	MPI, SPMD
External Package Dependencies	MPI
Mainly Used By	Academia
Related Works	N/A
Suggested By	UAB
Partner's Level of Expertise	Intermediate
Developed By	Department of Computer Architecture and Operating Systems at Universitat Autònoma de Barcelona
Main Referenced Publication	Roberto Solar, Remo Suppi, Emilio Luque, "High performance distributed cluster-based individual-oriented fish school simulation", Procedia Computer Science, Volume 4, 2011, Pages 76-85, ISSN 1877-0509, 10.1016/j.procs.2011.04.009
Version	1.0
License	Creative Commons
Website	N/A

3.3.5 HydroC

HydroC is an extract from the RAMSES application developed by the CEA Astrophysics division in Saclay, France. CEA's Romain Teyssier assembled HydroC to study the formation of large-scale structures and galaxies. Its following algorithms were simplified:

- The space domain is a rectangular two-dimensional splitting with a regular Cartesian mesh (there is no Adaptive Mesh Refinement);
- It solves compressible Euler equations of hydrodynamics;
- It is based on finite-volume numerical methods using a second-order Godunov scheme for Euler equations; and
- A Riemann solver computes the numerical flux at the interface of two neighboring computational cells.

HydroC has about 1,500 lines of C code and is ready to use on Unix systems.

The application is made of an iterative computation over time of the hydrodynamics of a perfect gas represented by a field of particles. HydroC uses a finite volume framework with a particle model build with two flux vectors. Each iteration is split in two; each iteration performs the Godunov computations on a particular dimension using a different order – a permutation.

The Godunov computation is composed of three steps:

- Computing primitive variables;
- Solving the Riemann problem at current cell interfaces, i.e. computing the Godunov state; and
- Computing incoming fluxes for horizontal or vertical interfaces from the Godunov state.

The ten different operations (implemented as C functions) used by HydroC are composed of a least one kernel. Over the ten operations, 22 loops execute over time and permutation for each iteration. The amount of computation performed by each kernel varies and is always parallel over at least one dimension (it can be a volume dimension of over flux variables). The most time-consuming operation is the Riemann Solver with the computation of the pressure, the density and the velocity for the fluxes. The main kernel loop is 140-line long and its asymptotic time complexity is in $O(n)$, where n is the grid size. HydroC contains an expensive loop using the Newton–Raphson method to find the appropriate accuracy for each iteration. During the computation, the expensive intrinsic square root is used 8 times, significantly increasing the volume of computation performed.

Despite its profusion of kernels and its solid background on physics simulation, HydroC remains easy to use and simple to validate. We provide a HMPP version tuned by a directive per loop defining the grid size on the GPU, thus offering at least one dimension to explore for auto-tuning. The major interest of the application is that there are many loops to tune; most loops have a close to flat profile on performance (except the Riemann one), considering our metric is the execution time.

The input data set is composed of a set of parameters that define the condition of the physics experiment. It is randomly filled using a set of parameters provided by a configuration file with the following properties:

- Time end: *tend* (by default, 40);
- Maximum number of time iterations: *nstepmax* (by default, 10);
- Mesh size: dimensions *nx* and *ny* with delta *dx* (by default, 6000x6000 *dx* 0.05);
- Mesh boundaries: *boundary_left*, *boundary_right*, *boundary_down*, *boundary_up* (by default 1x1x1x1); and
- Maximum number of iteration while solving the computation of precision in Riemann: *niter_riemann* (by default, 10).

This application has been described in the technical white paper "HYDRO", published by Guillaume Colin de Verdière from the CEA, and Pierre-François Lavallée, Philippe Wautelet, Dimitri Lecas, and Jean-Michel Dupays from IDRIS/CNRS in the context of the EU FP7 PRACE Project[13]. PRACE collaborators published several papers about the Hydro benchmark.[14]

As an initial test, the output and baseline performance figures shown in Figure 4 were gathered from an execution of HydroC on the Nova[15] cluster. The application was compiled using GCC 4.1.2 for this run.

[13] The Partnership for Advanced Computing in Europe (http://www.prace-ri.eu/).
[14] *"Final Software Evaluation Report", PRACE-1IP D9.2.2, (http://www.prace-project.eu/IMG/pdf/D9-2-2_1ip.pdf).*
[15] For a detailed description of the Nova cluster, refer to Appendix A: Specification of the reference machines.

```
+-------------------+
|nx=6000            |
|ny=6000            |
|tend=40.000        |
|nstepmax=10        |
|noutput=1000000    |
|dtoutput=0.000     |
+-------------------+
--> step=   1,  1.33631e-03, 1.33631e-03  {289.056 Mflops 7057177968}
(24.415s)
--> step=   2,  2.67261e-03, 1.33631e-03  {298.751 Mflops 7057181740}
(23.622s)
--> step=   3,  5.70914e-03, 3.03653e-03  {289.039 Mflops 7057186824}
(24.416s)
--> step=   4,  8.74568e-03, 3.03653e-03  {299.887 Mflops 7057190350}
(23.533s)
--> step=   5,  1.24942e-02, 3.74854e-03  {289.095 Mflops 7057192974}
(24.411s)
--> step=   6,  1.62428e-02, 3.74854e-03  {300.049 Mflops 7057195516}
(23.520s)
--> step=   7,  2.06309e-02, 4.38811e-03  {284.602 Mflops 7057197976}
(24.797s)
--> step=   8,  2.50190e-02, 4.38811e-03  {298.927 Mflops 7057200518}
(23.608s)
--> step=   9,  2.95700e-02, 4.55100e-03  {282.255 Mflops 7057202896}
(25.003s)
--> step=  10,  3.41210e-02, 4.55100e-03  {298.651 Mflops 7057205192}
(23.630s)
Hydro ends in 00:04:01.525s (241.525).
```

Figure 4: Results for HydroC compiled with GCC 4.1.2 and run on Nova

Short Name	**HydroC**
Full Name	A simple 2 Hydrodynamic Godunov in C
Purpose	A higher-order Godunov method for the radiation hydrodynamics
Area of Science	Hydrodynamics
Main Algorithms	Hydrodynamic Godunov
Total Lines of Code	2,596
Platforms	Linux workstations and clusters, NVIDIA GPUs
Language	C
Parallel Paradigm	OpenMP, HMPP
External Package Dependencies	HMPP 2
Mainly Used By	Industry
Related Works	http://www.itp.uzh.ch/~teyssier/Site/RAMSES.html
Suggested By	CAPS
Partner's Level of Expertise	Advanced

Developed By	Guillaume Colin de Verdière (CEA), Pierre-François Lavallée (IDRIS)
Main Referenced Publication	"Linearized formulation of the Riemann problem for radiation hydrodynamics", D. S. Balsara. Journal of Quant. Spect. and Radiative Transfer 61, pp 629-635, 1999
Version	1.0
License	CeCILL (CEA, CNRS, INRIA)
Website	N/A

3.3.6 NPB

The NAS Parallel Benchmarks (NPB) is a set of benchmarks or programs for performance evaluation of parallel supercomputers. These benchmarks are developed and maintained by the NASA Advanced Supercomputing (NAS) Division, which previously stood for the NASA Numerical Aerodynamic Simulation Program[16]. They are based on Computational Fluid Dynamics (CFD) applications and consisted of five kernels and three pseudo-applications (8 benchmarks in total) in the initial release specification (NPB1)[17]. NPB is one of the most important benchmarks in the scientific community and has a large visibility.

The current release of NPB contains the following benchmarks:

- Five kernels:
  - IS – Integer Sort: Sorts small integers using the bucket sort (exploits random memory access);
  - EP – Embarrassingly Parallel: Generates independent Gaussian random variates using the Marsaglia polar method;
  - CG – Conjugate Gradient: Estimates the smallest eigenvalue of a large sparse symmetric positive-definite matrix using the inverse iteration with the conjugate gradient method as a subroutine for solving systems of linear equations (exploits irregular memory access and communication);
  - MG – Multi-Grid on a sequence of meshes: Approximates the solution to a three-dimensional discrete Poisson equation using the V-cycle multi-grid method (exploits long- and short-distance communication and memory intensiveness);
  - FT – discrete 3D Fast Fourier Transform: Solves a three-dimensional partial differential equation (PDE) using the Fast Fourier Transform (FFT) (exploits all-to-all communication).
- Three pseudo-applications that solve a synthetic system of nonlinear PDEs using 3 different algorithms for the solver kernel: block tridiagonal (BT), scalar pentadiagonal (SP) and symmetric successive over-relaxation (SSOR).
  - BT – Block Tridiagonal solver;
  - SP – Scalar Pentadiagonal solver;
  - LU – Lower-Upper Gauss-Seidel solver.

NPB was enlarged to include new benchmarks for unstructured adaptive mesh, parallel I/O, multi-zone applications, and computational grids.

- Multi-zone applications:
  - BT-MZ – uneven-size zones within a problem class, increased number of zones as the problem class grows;

[16] http://www.nas.nasa.gov/publications/npb.html
[17] *"The NAS Parallel Benchmarks 2.0", D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, M. Yarrow. NAS Technical Report NAS-95-020, NASA Ames Research Center, Moffett Field, CA, December 1995*

  - SP-MZ – even-size zones within a problem class, increased number of zones as the problem class grows;
  - LU-MZ – even-size zones within a problem class, a fixed number of zones for all problem classes.
- Unstructured computation, parallel I/O, and data movement:
  - UA – Unstructured Adaptive mesh (exploits dynamic and irregular memory access);
  - BT-IO – Test of different parallel I/O techniques;
  - DC – Data Cube;
  - DT – Data Traffic.
- Computational grids:
  - ED – Embarrassingly Distributed;
  - HC – Helical Chain;
  - VP – Visualization Pipeline;
  - MB – Mixed Bag.

Older benchmarks were used before NPB, but they were designed for vector computers. They were also not suitable for parallel systems, due to their parallelism-impeding tuning restrictions and insufficient problem sizes.

Problem sizes in NPB are defined in classes:

- Class S: small for quick test purposes;
- Class W: workstation size (a 90's workstation, now likely too small);
- Classes A, B, C: standard test problems (~4x size increase going from one class to the next); and
- Classes D, E, F: large test problems (~16x size increase from each of the previous classes).

Reference implementations of NPB are available in MPI and OpenMP (NPB 2 and NPB 3).

Short Name	**NPB**
Full Name	NAS Parallel Benchmarks
Purpose	A small set of programs to help evaluate the performance of parallel supercomputers
Area of Science	Computational fluid dynamics
Main Algorithms	N/A
Total Lines of Code	N/A
Platforms	Linux platforms and all HPC systems
Language	Fortran, C
Parallel Paradigm	MPI, OpenMP, Hybrid
External Package Dependencies	MPI
Mainly Used By	Benchmarkers, tool developers
Related Works	SPEC OMP, SPEC MPI, HPC Challenge
Suggested By	TUM
Partner's Level of Expertise	Intermediate
Developed By	NASA Advanced Supercomputing (NAS) Division

Main Referenced Publication	"The NAS Parallel Benchmarks 2.0". D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, M. Yarrow, NAS Technical Report NAS-95-020, NASA Ames Research Center, Moffett Field, CA, December 1995
Version	3.3
License	NASA Open Source Agreement (NOSA)
Website	http://www.nas.nasa.gov/publications/npb.html

3.3.7 SeisSol

SeisSol is an MPI-parallel Fortran90 application developed by the Department of Earth and Environmental Sciences at the Ludwig-Maximilian University. The application is used for simulating realistic earthquake scenarios, accounting for a variety of geophysical processes that affect the propagation of seismic waves – e.g. viscoelastic attenuation, strong material heterogeneities and anisotropy.

The accurate numerical simulations of the propagation of seismic waves help in understanding complex wave phenomena. SeisSol includes capabilities to adapt to complex 3D geometries by using tetrahedral and hexahedral meshes. It also can handle acoustic, elastic, viscoelastic, poroelastic and anisotropic material properties to approximate realistic geological sub-surfaces. The internal algorithms use arbitrarily high-order approximations in time and space as well as an explicit local time stepping, such that each element runs with its own optimal time-step length to reduce computation time.

The computational kernel performs a global time marching, in which each sub-domain is treated. The subdomains are generated with the same number of cells, and calculations done to each subdomain for the smallest time step are communicated. The sub-domains themselves perform internally a local time marching.

Given that SeisSol is a scientific code intensive in arithmetic operations, it is an ideal candidate for energy optimization. One can apply Dynamic Voltage and Frequency Scaling (DVFS)[18] and expect that the code will have an optimal frequency within the midrange of frequency values.

Our most relevant input data set allows SeisSol to simulate a slice of the Earth's crust using almost 4 million cells. The METIS software package[19] aids us generating the mesh.

The application was run in a 16-core node on SuperMUC. Execution flags include the *O3* optimization for loops with intensive arithmetic operations. The averaged result for 5 runs of an Earth's crust simulation was 36,228 Joules with an execution time of 435 seconds.

Short Name	**SeisSol**
Full Name	Seismic Wave Propagation Solutions for Realistic 3D Media
Purpose	SeisSol provides wave propagation solutions in an elastic medium in 3D with geometrically complex domains
Area of Science	Geophysics
Main Algorithms	Discontinuous Galerkin Finite Element Method, Cauchy-Kovalewski
Total Lines of Code	~50,000
Platforms	Linux, Altix 4700, SuperMIG, BlueGene
Language	Fortran

[18] For the definition and rationale behind the use of Dynamic Voltage and Frequency Scaling, refer to Report D4.1, "Design of the Tuning Plugins".
[19] METIS is a set of serial programs for partitioning graphs, partitioning finite-element meshes, and producing fill-reducing ordering for sparse matrices. It is available at http://glaros.dtc.umn.edu/gkhome/views/metis.

Parallel Paradigm	MPI
External Package Dependencies	MPI
Mainly Used By	Academia, research
Related Works	N/A
Suggested By	LRZ
Partner's Level of Expertise	Intermediate
Developed By	Department of Earth and Environmental Sciences at the Ludwig-Maximilian-University Munich
Main Referenced Publication	http://www.geophysik.uni-muenchen.de/~kaeser/SeisSol/publications.html
Version	SeisSol 5 Benchmark Periscope Version
License	N/A
Website	http://www.geophysik.uni-muenchen.de/~kaeser/SeisSol/

3.3.8 Sip

Sip (Strongly implicit procedure) is written in Fortran and parallelized using OpenMP. This application implements a method suitable for solving systems of linear equations resulting from the discretization of partial differential equations. Its method is widely used in fluid mechanics and therefore of great importance.

Since this application does not make use of MPI, Sip cannot be used to test MPI-related plugins. Nonetheless, due to the different features this application provides in comparison with other applications available in the application repository, Sip was chosen as a test bench for energy optimization.

The current benchmark is intended to test the saturated single-node performance and the scaling behavior. Regarding the implementation, different approaches are possible. There are currently the 6 following versions implemented:

- Case 101 (SipThreeDSolver): iterates over all nodes in 3 do-loops (straightforward approach);
- Case 102 (SipThreeDSolver_regopt): same as 101 but one loop is split in multiple parts to improve register usage;
- Case 103 (SipThreeDSolver_ppp, pipeline parallel processing): data dependencies may prohibit automatic parallelization of the 3D-Version of the Sip-solver;
- Case 104 (SipHyperplaneSolver): considering the LU-Decomposition of an hyperplane, calculations on data within a plane are independent of each other and can be parallelized;
- Case 105 (SipHyperplaneSikver_sr8k): pipeline-parallel variant of case 104 optimized for Hitachi SR8000; and
- Case 106 (SipHyperLineSolver): Similar to hyperplanes one can also define hyperlines and apply them to solving the SIP.

Sip's binary takes the input data from the *SipInput.dat* file. This file contains, as an example, the values shown in Figure 5.

```
     case stride iter lengths
     103 1 1 301 0 0 0 0 0 0
```

Figure 5: Sample input for running Sip's case 103

Should it be run with the input depicted in Figure 5, Sip would execute the 103-version algorithm over a matrix of 300 elements. For running the 6 algorithms, an input such as the one in Figure 6 is needed.

```
     case stride iter lengths
     101 1 1 301 0 0 0 0 0 0
     102 1 1 301 0 0 0 0 0 0
     103 1 1 301 0 0 0 0 0 0
     104 1 1 301 0 0 0 0 0 0
     105 1 1 301 0 0 0 0 0 0
     106 1 1 301 0 0 0 0 0 0
```

Figure 6: Sample input for running all Sip's 6 cases

The output for the sample input file described in Figure 5 is shown in Figure 7.

```
Processes:      1
Threads:        2
Result Filename:     SipB ench0001_002.res
rinf2_init: Starting benchmark. Please, wait...
rinf2_init: Overhead is   3.186702728271484E-009 seconds.
(103) SipThreeDSolver_ppp
veclen stride iter. time/iter. avg. Perf. maxdev. avg. BW block secs MFlop/s
%MByte/s
301 1 1 1.726E+02 2.387E+02 0.00 1.791E+03
Performance per core (Mflop/s): 119.3
```

Figure 7: Sip's output for an execution of case 103

To obtain the results presented in Figure 7, Sip was run using 8 cores of a SuperMUC node. The average performance for 5 runs of the same data set was 14,380.9 Joules with an execution time of 393 seconds.

Short Name	**Sip**
Full Name	Strongly-Implicit Procedure
Purpose	Solving systems of linear equations resulting from the discretization of a partial differential equation
Area of Science	Widely used in Fluid Mechanics
Main Algorithms	LU Decomposition
Total Lines of Code	< 5,000
Platforms	Linux, Altix 4700, SuperMIG

Language	Fortran 90
Parallel Paradigm	MPI
External Package Dependencies	MPI
Mainly Used By	Academia, research centers
Related Works	http://www.rrze.uni-erlangen.de/dienste/arbeiten-rechnen/hpc/Projekte/OptGuide.pdf
Suggested By	LRZ
Partner's Level of Expertise	Intermediate
Developed By	HPC Group of Regionales Rechenzentrum Erlangen
Main Referenced Publication	"Strongly Implicit Procedure - SipBench", Peridot Performance Workshop, 2002
Version	1.0-23.01.2003
License	N/A
Website	http://www10.informatik.uni-erlangen.de/~deserno/SipBench.pdf

3.3.9 S2F2M

The main goal of the S2F2M statistical simulator is to predict the propagation of wild fires and thus provide useful information to firefighters and the support personnel. This application was developed by the CAOS department from UAB. The simulator is intended for use onsite to predict patterns of fire propagation in real time and to help decide evacuation routes or firefighting tactics.

The time constraints and the difficulty of obtaining real data from the fire make this kind of simulation very complex; one must first determine the conditions of the terrain and atmospheric parameters before being able to correctly predict spreading patterns. S2F2M uses a genetic algorithm to find the best fitted sets of parameters and uses a statistical model to combine the results obtained into the most probable outcome.

S2F2M must run thousands of simulations in order to get accurate results. This requires a lot of time and thus a parallel approach has been devised to speed up the process. It uses the Master-Worker paradigm to distribute the workload, in the form of different scenarios to be simulated, among the available workers. Once each worker finishes its set of simulations it sends back the results to the Master and is ready to compute the next set of scenarios. The fire simulations are carried out by the fireLib library[20].

This application's characteristics make it suitable for Master-Worker plugin, since it can tune the partitioning and distribution of the load and the number of workers used. The application's input is composed of the initial conditions of the fire (i.e. the cells initially on fire), the topographic details of the terrain, and the weather conditions in the form of configuration files.

Short Name	**S2F2M**
Full Name	Statistical Model of Fire Simulator
Purpose	Simulate the propagation of forest fires to provide useful information, such as evacuation possibilities
Area of Science	Simulation
Main Algorithms	N/A

[20] fireLib is a C function library for predicting wildland fire behavior using the BEHAVE algorithms (http://fire.org/).

Total Lines of Code	~6,000
Platforms	Linux clusters
Language	C
Parallel Paradigm	Master-Worker (MPI)
External Package Dependencies	MPI
Mainly Used By	Academia
Related Works	"fireLib User Manual and Technical Reference". C. D. Bevins, 1996
Suggested By	UAB
Partner's Level of Expertise	Intermediate
Developed By	Department of Computer Architecture and Operating Systems at Universitat Autònoma de Barcelona
Main Referenced Publication	"Towards Dynamic Data Driven Forest Fire Propagation Prediction", Roque Dario Rodriguez Aseretto. PhD thesis, Universitat Autònoma de Barcelona, 2010
Version	1.0
License	Creative Commons
Website	N/A

3.4 The Application Repository at a Glance

In Table 4 we present a summary of the 9 applications we initially selected to compose our repository, along with their characteristics and the plugins that can benefit from it. Green cells point to characteristics the applications display, whereas red cells indicate an application does not exhibit the characteristic.

		Repository Applications								
		BLAST	Convolution	FaceDetect	FSSIM	HydroC	NPB	SeisSol	Sip	S2F2M
Plugins	High-level parallel patterns for GPGPU	F	F	T	F	F	F	F	F	F
	Hybrid manycore HMPP codelets	F	T	F	F	T	F	F	F	F
	Energy efficency via CPU frequency	T	F	F	T	T	T	T	T	T
	Master-worker MPI	T	F	F	F	F	F	F	F	T
	MPI runtime	T	F	F	T	F	T	T	F	T
	Compiler flag selection	T	F	F	T	T	T	T	T	T
Languages	C	T	T	T	F	T	T	F	F	T
	C++	T	F	T	T	F	F	F	F	F
	Fortran	F	F	F	F	F	T	T	T	F
Parallel technology	OpenMP	F	T	F	F	T	T	F	T	F
	MPI	T	T	F	T	T	T	T	T	T
	OpenCL	F	T	F	F	T	F	F	F	F
	HMPP	F	T	F	F	T	F	F	F	F
	CUDA	F	T	T	F	T	F	F	F	F
	OpenACC	F	T	F	F	T	F	F	F	F
Target Systems	Machines without GPUs	T	T	T	T	T	T	T	T	T
	Machines with GPUs	F	T	T	F	T	F	F	F	F
	SuperMUC	T	F	F	T	F	T	T	T	T
Type	Benchmark	F	T	F	F	F	T	F	T	F
	Mini-application	F	F	T	F	T	T	F	F	F
	Real application	T	F	F	T	F	F	T	F	T

Table 4: Applications in the repository's first installment, their characteristics and plugins benefitted

We can gather from Table 4 that we provide a good coverage of the tuning plugins and techniques with the repository we assembled. A total of 7 out of 9 applications can compose test cases for the plugins for both compiler flag selection tuning and energy consumption tuning via CPU frequency, due to their loose input requirements. The master-worker pattern is present in 2 of the applications, which can assess the plugin for MPI master-worker pattern tuning. HMPP directives are also present in two of the applications, which can be used to evaluate the plugin for HMPP codelet tuning. The MPI parallel technology is the most widespread one within our applications and hence the MPI runtime tuning plugin can take advantage of a total of 5 out of 9 applications for feedback during the design, implementation and integration steps.

We aimed to have every plugin covered by at least one application and we managed it. Indeed, apart from the plugin for tuning of high-level parallel patterns for GPGPU, all other plugins have at least 2 applications to provide them with feedback. This plugin requires applications written in CUDA or OpenCL using the pipeline pattern, which is a very promising approach for writing scientific applications that is now becoming widespread. In spite of its topicality and originality, the number of applications that presently display these combined features limits our approach. Nonetheless, as the project progresses, many more real-world applications will fit these characteristics and the repository is expected to account for them in due time.

It should also be noted that we selected at least 3 applications according to the application types we characterized our population in (benchmark, mini-application, real application). Each type provides a distinct use case within the evaluation process (e.g. how generic and reusable the results for tuning are, how comparable they are to other results) and the combination of different types provides a broad view of the impact the tuning techniques and plugins have on scientific applications.

Overall, we believe the repository we have assembled displays a very good coverage of all the application characteristics, plugins and techniques to be assessed. Given all the applications selected fulfill the desirable features we envisaged, the repository has become a powerful tool for the evaluation and assessment tasks within the AutoTune project.

4 Manual Tuning

The manual tuning of applications is a very important step in steering and assessing the tuning techniques. This laborious activity is able to provide first insights into the approach and practical implementation of the tuning plugins. More importantly, a first manual tuning represents proof of concept that a tuning technique works in practice. This is an essential milestone to strengthen the rationale for why investing in plugin development is a worthy task.

The primary reason behind automating such manual tuning techniques is the inherent human dependency. Not only are humans more prone to errors, but they also take longer to query and analyze an application, both of which lead to a longer time effort spent on tuning. In this Chapter we show the steps needed to gather information, compose a search space, and manually tune an application. The complex task of manually tuning applications proves to be a rather quicker task when performed automatically.

For our manual tuning, we selected one application per tuning technique. Besides proving the point that the tuning technique is effective, each application performs a smoke testing[21] on the tuning technique. The difference between these is that whereas the proof of concept evaluates the outcomes to judge whether the technique is effective, the smoke testing analyzes the effectiveness of each of the steps that compose the technique. Both approaches combined increase the user and developer confidence in the technique.

In this Chapter we discuss the employment of our tuning techniques to manually tune a set of applications. We begin by introducing the procedure we used to tune the applications, and then follow with a description of the steps taken before and during the tuning. We finalize the Chapter with an analysis of the improvements obtained by the manual tuning, along with the effort in time taken for the tuning.

[21] Smoke testing aims at revealing simple failures severe enough to reject a prospective approach or technique. A test case that covers the most important functionalities is executed to ascertain that the most crucial steps work correctly.

4.1 The Tuning Procedure

Currently, when the search space is small, the procedure to search for the best code variant is exhaustive. All the variants are outlined and tested, and the variant that displays the best figures for the performance metrics is elected. When the search space is large, however, heuristics such as the Bisection Method are applied.[22]

As the developers are currently performing the manual tuning, they are also performing the traversal within the search space. In order to reduce the time taken on the manual tuning procedure as much as possible, the tuning developers gather inputs from the application from several sources, including but not limited to application benchmarks, profiling and tracing tools. The use of external tools neither jeopardizes the "manual" characteristic of the tuning nor biases the tuning procedure, since the output gathered by such tools can still be acquired by the tuning plugin upon automatic tuning.

It should be noted that the manual tuning procedure here undertaken is the same implemented by the tuning plugins. Our objective is to assess the practicability of employing such tuning techniques to real applications and achieving good performance gains.

4.2 Approach and Results

In this Section we describe the procedures and results for the manual tuning of repository applications. The manual tunings were conducted as proofs of concept that their respective tuning techniques are effective and efficient. Therefore, we divided this Section into subsections, one per tuning plugin, where we prove the technique worthwhile, along with the following points:

- Description of the efficiency issues within the application to be manually tuned;
- Results of the metric measurements (energy efficiency, execution time) before tuning (baseline figures);
- Depiction of the approach taken for tuning;
- Results of metric measurements for each dataset proposed after tuning;
- Comparison between the metric measurements before and after tuning; and
- Time effort taken for the manual tuning.

We collected results of the metric measurements exclusively during the execution of the application's computational kernels of interest. This approach was implemented to avoid the influence of the pre- and post-processing tasks, which are mostly serial and do not benefit from the tuning techniques. The pre- and post-processing times may become dominant by the time the computational kernels of interest are optimized, and hence eclipse the improvements there achieved.

This Section covers the manual tuning of repository applications as proofs of concept that the tuning techniques are worthwhile. The analysis is based on performance figures taken from the execution on reference machines, as well as on time effort spent by the developers. For an introductory discussion about each application's features and behavior, refer to Chapter 3.

4.2.1 Tuning of High-Level Parallel Patterns for GPGPU

We consider auto-tuning of high-level parallel patterns that are realized based on while-loops and corresponding annotations in the source code. From within such a pipeline, the underlying runtime system calls construct functions (realized as components) and dynamically schedules them to the execution units of single-node heterogeneous manycore architectures (e.g. a CPU/GPU-based system).

Our primary objective of tuning such pipelined applications is to maximize the pipeline throughput. Since we employ GPUs as well as CPUs, there are several layers to consider. Initially we will focus on tuning the parameters related to the main characteristics of the pipeline, most notably the stage replication factor.

[22] A more thorough description of the tuning techniques can be found in Document D4.1, "Design of the Tuning Plugins".

Increasing the replication factor of a stage may mitigate the effect of stages with relatively high execution times, since several stage instances will be executed in parallel, resulting in increased pipeline throughput. Other potential tuning options may be considered at the level of the runtime system, where different scheduling policies, as well as the parameters related to the hardware characteristics such as number of CPUs and GPUs, may be valuable candidates for tuning. We investigate two versions of the application: the baseline OpenCV version, which performs the face detection in the described stages; and the pipelined version developed on top of the StarPU runtime system.

Table 5 shows the measurements for an experiment that performs face detection on “Image dataset 1” described in Section 3.3.3. We used same pre-trained model (classifier) “*haarcascade_frontalface_alt.xml*” for both the baseline and the manually tuned version. Performance results included were obtained by averaging the execution time of 5 runs. For comparison, we also include the results achieved with a baseline OpenCV version with and without TBB support. The hardware in use is a machine with two quad-core Intel Xeon X5550 CPUs at 2.66GHz and 24GiB DDR3-1333 of system memory, equipped with two NVIDIA Tesla C2050 GPUs and one NVIDIA Tesla C1060 GPUs.

Baseline version (OpenCV)	**Average execution time (in seconds)**
With GPU support	101.6135
With TBB	169.5282
Without TBB	1,249.6688

Table 5: OpenCV’s baseline runtime figures for FaceDetect

The results in Table 6 demonstrate the impact of various parameter sets used to evaluate the application’s pipelined version. Stage replication factors shown here are applied to all three stages. Although the HEFT (heterogeneous earliest finish time) scheduling policy relies on historic performance data and the results generally improve after repeating the execution of the experiment on the same hardware, results show that tuning stage replication factors can be beneficial on particular combination of hardware characteristics, potentially bringing improvement in performance. On the other side, the EAGER scheduling policy, proved to be significantly slower in general, even compared to the baseline OpenCV implementation with GPU support. However, the difference becomes less significant when stage replication factors are adjusted. The overhead of the StarPU runtime, shown in each right column per tested architecture, increases with the number of GPUs in use. However, it does not depend on the size of the dataset, hence the overhead becomes insignificant when the dataset is sufficiently large.

Scheduling policies and replication factors		**Runtime without (left column) and with (right column) the StarPU overhead**					
		7 CPUs + 1 GPU		6 CPUs + 2 GPUs		5 CPUs + 3 GPUs	
HEFT	4	74.333	83.483	56.163	75.409	56.365	80.647
	10	74.244	83.322	56.064	75.702	52.941	77.007
	20	63.790	72.966	53.049	72.350	50.998	75.148
	50	74.768	84.165	52.807	71.990	51.686	76.025
EAGER	4	240.168	249.426	225.263	244.653	214.208	238.559
	10	111.345	120.394	104.813	124.161	104.373	128.888
	20	112.468	121.485	103.053	122.337	98.805	123.358
	50	109.578	118.698	102.329	121.705	99.464	123.336

Table 6: FaceDetect’s runtime per scheduling policy, replication factor and hardware resources

Finally, the time effort invested in tuning includes several different aspects:

- Developing the baseline and the basic pipelined versions of the application: ~32 man-hours;
- Optimizing the pipelined version with respect to the runtime: ~40 man-hours;
- Preparing the data set: ~8 man-hours; and
- Running and evaluating the experiments with different set of parameters: ~40 man-hours.

4.2.2 Tuning of Hybrid Manycore HMPP Codelets

The tuning of the HMPP codelets was experimented using the Convolution application, which offers an interesting set of tuning points for HMPP applications. The tuning objective is to obtain the lowest execution time with different parameter values for the loop transformations and the grid directives.

As a baseline we compiled Convolution using the ICC 12.1 compiler and executed on one of Nova's 8 cores, on five images with a default expansion factor of 8. An analysis of execution time of the "Push+Blur" stencil is in Table 7.

Image	**Code Variant**		**Speed-Up**
	Sequential	OpenMP	
Eusebius@commonsSaint_Malo-CC_BY_2.0.tif	3.081	0.523	5.894
Francois_Bodin-Paysage01-CC_BY_2.0.tif	14.143	1.864	7.587
Francois_Bodin-Paysage02-CC_BY_2.0.tif	9.233	1.270	7.269
Francois_Bodin-Paysage04-CC_BY_2.0.tif	14.712	1.949	7.548
Michal-Osmenda-Mont_Saint_Michel-CC_BY_SA_2.0.tif	2.066	0.386	5.352

Table 7: Baseline runtime figures for the Push+Blur stencil for expansion factor 8 on a Nova node

From the original application, we performed a port to HMPP and optimized the data transfers from the CPU memory to the GPU memory. For the manual tuning, we focused on the exploration of unrolling factors that could be applied to the loop nest inside the convolution kernel. Figure 8 provides a piece of code developed during the HMPP port; the tuning parameters are enclosed between angle brackets.

```
#pragma hmppcg unroll <Unrolling Factor>, jam
     for (i = stencil; i < heigh - stencil; i++) {
          for (j = stencil; j < width - stencil; j++) {
                ...
          }
     }
```

Figure 8: A port of the convolution kernel to HMPP

The tuning of the HMPP parameters commonly takes between 30 minutes and 1 hour for a single input data set, depending on the expertise of the developer. Table 8 presents the results we produced.

Code Variant	Unrolling Factor	Runtime (s)	Speedup
Sequential	N/A	2.066	0.19
OpenMP (8 cores)	N/A	0.386	1.00
HMPP	0 (none)	0.786348	0.49
HMPP	2	0.489006	0.79
HMPP	4	0.345937	1.12
HMPP	8	0.266592	1.45
HMPP	16	0.235939	1.64
HMPP	32	2.18477	0.18

Table 8: Runtime for Convolution per code variant and unrolling factor

From this experiment, we demonstrate that the tuning provided a significant improvement on the kernel execution time, up to a factor of 1.6 compared to the 8-core OpenMP version. We acknowledge that these speed-up figures refer to an execution on a GPGPU from an old generation, as well as that they do not take data transfers into account (however negligible in our experiment).

Note that for a real application the same procedure has to be done for all loop nests and for different types of loop transformations (e.g. tiling, permutation, grid size). Thus, considering the tuning effort, the work performed in Convolution relies on an implicit high-user expertise. For example, the loop transformation Unroll-and-Jam performed is a typical optimization technique an inexperienced user would struggle to implement; this is only one among all transformations available for HMPP.

Using a naïve approach, we decompose a potential tuning effort as the following:

- Exploring the grid size directive: 1 dimension of minimum size 5 (512x1 to 32x16);
- Exploring permutations: 1 dimension of size 2;
- Unrolling, 2 dimensions (one by loop) multiply by:
    - 1 dimension of size 7 for the unroll factor (0 to 32 and full-unroll);
    - 1 dimension of size 2 for the jam option;
    - 1 dimension of size 2 for the split option;
    - 1 dimension of size 3 for the remainder policy (none, guarded, no-remainder).

Only with these parameters (disregarding tiling, loop fusion and loop distribution, which are less frequently useful), the manual tuning implies a *5x2x(2x(7x2x2x3)) = 1680* combinations per kernel. This accounts for about 140 man-hours if the developer can test each different combination every 5 minutes.

4.2.3 Tuning of Energy Consumption via CPU Frequency

Manually tuning the CPU frequency controlled from within the code segments is extremely time-consuming without any additional tool developed for this purpose. We used a library to gather a rough idea about how to tune. The library alleviates several problems incurred in measuring the RAPL counters[23], such as the overflows of counters that occur every 60 seconds due to hardware constraints. We were able to partially tune SeisSol, establishing tuning parameters per so-called code regions, as defined by Periscope.

Table 9 contains some of the Periscope regions we coded, given their interest to SeisSol.

[23] For a detailed explanation about RAPL counters and frequency policies, refer to Report D4.1, “Design of the Tuning Plugins”.

Region ID	Meaning
1	Main
2	Sub, call, user
3	Loop, for, forall
8	I/O
10	Parallel

Table 9: Periscope regions coded for SeisSol

For the first tests, we defined seven possible scenarios that we tuned per region type. We know that a same type of region might behave differently and therefore frequency policies could be set depending on the segment code and not the region type. However, we lack the automatic tools to do the optimization per code segments and 100% manual tuning would be extremely time-consuming. The manual tuning is thus currently limited, although there is tuning potential for frequency scaling per code segment. The scenarios T1 to T7 are defined in Table 10.

Optimization Profile (scenario)	Periscope Region Type				
	1	2	3	8	10
T1	ondemand 2.7	ondemand 2.7	ondemand 2.7	ondemand 2.7	ondemand 2.7
T2	userspace 2.7	userspace 2.7	userspace 2.7	userspace 2.7	userspace 2.7
T3	ondemand 1.9	ondemand 1.9	ondemand 1.9	ondemand 1.9	ondemand 1.9
T4	userspace 1.2	userspace 1.2	userspace 1.2	userspace 1.2	userspace 1.2
T5	ondemand 1.9	ondemand 1.9	ondemand 2.7	ondemand 2.7	ondemand 2.7
T6	ondemand 1.9	ondemand 1.9	userspace 1.9	ondemand 2.7	userspace 1.2
T7	ondemand 1.9	ondemand 1.2	userspace 1.2	ondemand 1.2	userspace 1.2

Table 10: Scenario variants generated for SeisSol

Table 11 shows the performance results for SeisSol tuned on SuperMUC[24]. As previously mentioned these results may not necessarily be the optimal result available. We believe automatic tuning can optimize the energy consumption even further.

[24] For a detailed description of SuperMUC, refer to Appendix A: Specification of the reference machines.

Profile	Energy (J)	Time (s)	Energy (J/CPU)
T1	35,713.6	473.14	1116.05
T2	29,269.0	387.81	914.66
T3	26,478.5	543.21	827.45
T4	26,060.3	835.65	814.38
T5	37,001.5	496.71	1156.3
T6	24,523.6	513.44	766.36
T7	24,622.8	774.01	769.46

Table 11: Performance figures per optimization profile for SeisSol on SuperMUC

Note that the best scenario is the T6, once it consumes the least amount of energy among all scenarios. However, scenario T2 spends significantly less time and consumes slightly more energy compared to T6.

Our tuning efforts can be split into the following:

- Manually tuning the applications: ~16 man-hours;
- Compiling and linking the application, preparing scripts and running the first tests: ~16 man-hours;
- Running tests, gathering results and comparisons to understand the relation between the optimizations performed and the behavior observed: ~80 man-hours.

4.2.4 Tuning of Master-Worker MPI

Master-Worker MPI applications are optimized by tuning MPI and application-specific parameters such as the partitioning factor of the workload and the amount of workers being executed. The objective here is to balance the load among the workers.

We chose BLAST as the subject of study for the Master-Worker MPI tuning. The following table contains the execution times in seconds for query alignments in BLAST of Input A (1-MiB query) over the partitioned database with 128 fragments.

Workers	Expected runtime (s)	Real runtime (s)	Error (%)
2	38,375.81	38,410.30	0.09
4	19,136.69	19,261.70	0.65
8	9,584.92	9,620.99	0.37
16	4,798.99	5,242.65	8.46
32	2,409.03	5,264.33	54.23

Table 12: Expected and obtained runtime for BLAST for Input A

In Table 12 we compare BLAST's expected runtime performance with different number of workers for the same workload – a database partitioned in 128 fragments. We expect an almost linear reduction in execution time as we use more resources. We see that when using 16 workers the performance starts to deviate from our expectations; increasing the number of workers to 32 further expands this expectation gap.

Figure 9 depicts the situation for different partition factors.

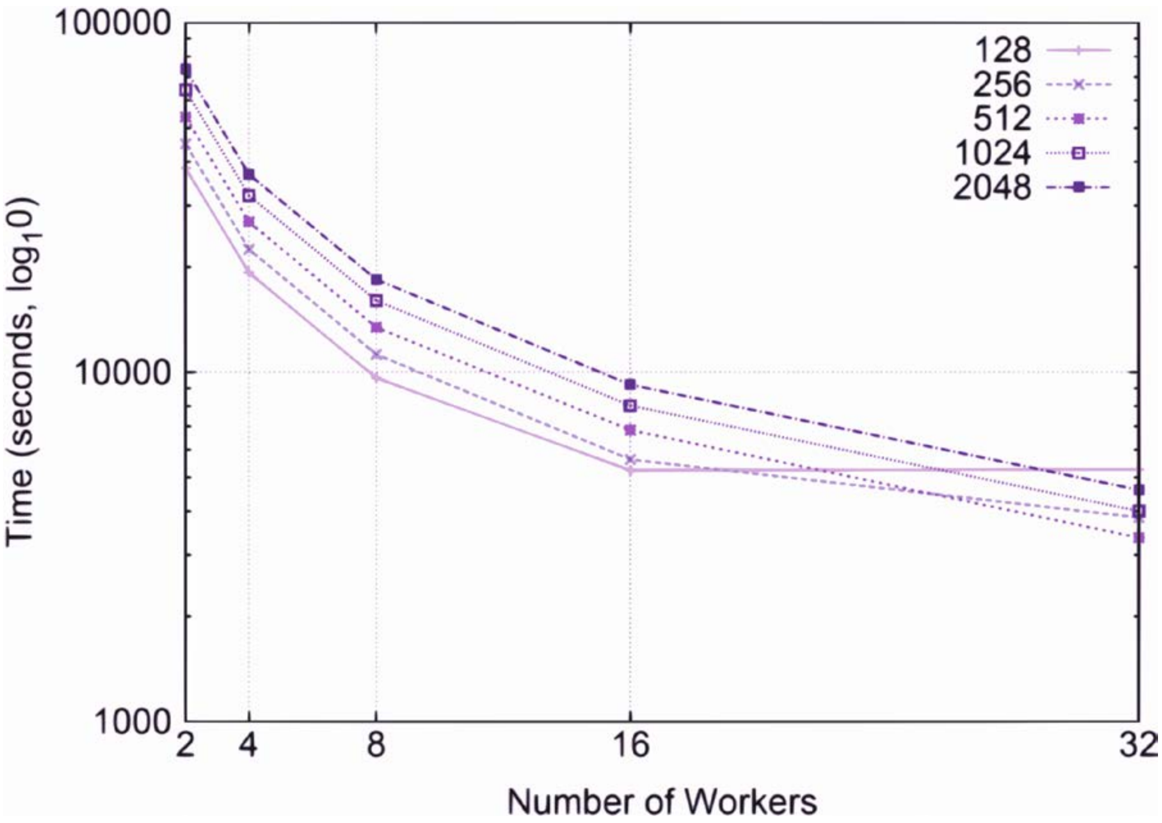


Figure 9: Runtime for BLAST for different number of workers and partition factors

For BLAST running with over 32 workers, the partition factors of 128 and 256 are too low. This indicates that the granularity with which we distribute the load among workers is not fine enough.

The time taken for the manual tuning can be divided as it follows:

- Studying the application, finding relevant tuning points and evaluating their impact on performance: ~56 man-hours;
- Preparing the different partitions of the workload and the scenarios to run: 32 man-hours;
- Running scenarios: 24 man-hours; and
- Evaluating results: 8~16 man-hours.

4.2.5 Tuning of MPI Runtime

MPI libraries allow us to configure the behavior of the Message Passing Interface library, providing a way for expert users to adapt their applications to the underlying architecture. When these MPI options are configured to suit a particular kind of application running on a specific architecture, the application performance can be greatly increased. However, manually tuning these options is very complex. By applying automatic searches, the MPI Runtime plugin is able to automatically detect which options have a relevant impact in the application execution and tune them accordingly to achieve the best possible performance.

Some relevant MPI options are outlined as it follows:

- MP_TASK_PER_NODE;
- MP_TASK_AFFINITY;
- MP_BUFFER_MEM;
- MP_EAGER_LIMIT.

Certain changes to the structure of the MPI directives or use of different communication techniques (e.g. blocking and non-blocking communication) can also improve the overall application performance by overlapping computation and communication. Moreover, load balancing can be applied, in a subtler form, to all MPI applications regardless of the paradigm.

The manual tuning for testing MPI configuration was carried out using the FSSIM application, which performs intense message passing.

Eager Limit	**Buffer Size**	
	32 MiB	64 MiB
1024 B	413.172 s	413.650 s
8192 B	451.824 s	455.807 s
32,768 B	483.243 s	505.693 s
65,536 B	522.651 s	526.995 s

Table 13: Runtimes for FSSIM for different eager limits and buffer sizes

Table 13 shows the time in seconds for different configurations of the eager limit (given in bytes) and the buffer size (given in MiB). The experiments consist of runs of the FSSIM simulator on 92 cores, performing 100 simulation steps over the medium-sized input dataset. Only sample values were tried for the MPI options because performing a manual exhaustive search is prohibitively time-consuming.

Figure 10 graphically presents the data on the table. We can appreciate that given the characteristics of the application the best configurations are those with lower eager limit.

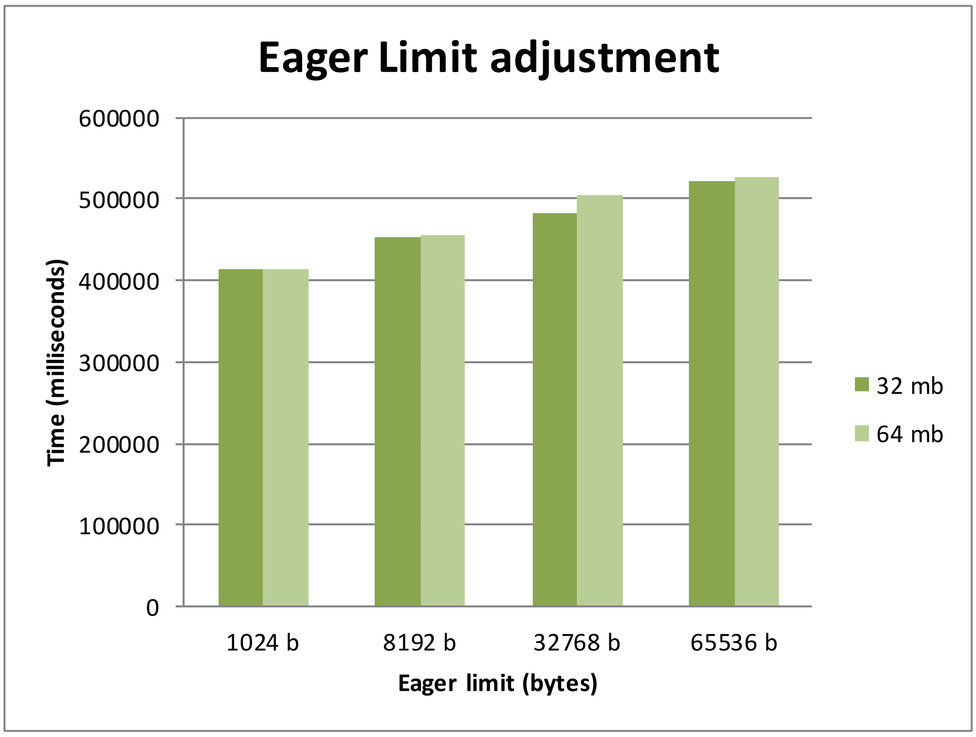


Figure 10: Runtime figures for FSSIM for different eager limits and buffer sizes

In our experiments, the buffer size had no relevant impact on performance, although the 64-MiB configuration yields slightly slower execution times. Nevertheless, the buffer size may be an important tuning point in on other systems with memory constraints.

Moreover, FSSIM was further optimized by changing the code structure to improve the passing of messages. Changing the blocking calls of *MPI_send* and *MPI_recv* in the communication between processes permitted the overlap of communication and computation, thus reducing the application's total wall time.

Workload	Original exec. time	Optimized exec. time	Speed-up
131072	248.7	176.2	1.41
262144	594.7	502.0	1.18
524288	1741.9	1582.5	1.10

Table 14: Runtime figures for FSSIM before and after tuned for SuperMIG

The results presented in Table 14 correspond to executions of FSSIM on SuperMIG with different workloads. FSSIM ran for 100 iterations in 160 cores.

The time dedicated for the tuning is divided as it follows:

- Evaluating the impact of the MPI options and generating significant sample values: 32 man-hours;
- Preparing the application and scripts for execution:
    - Configuring the MPI: 16 man-hours;
    - Coding the optimizations on the MPI structure: 32 man-hours;
- Running tests: 24 man-hours; and
- Evaluating results: 8~16 man-hours.

4.2.6 Tuning of Compiler Flag Selection

The objective of this tuning is to improve the execution time of the NPB applications (BT, SP and LU) by compiling them with a selection of compiler flags that improve their execution time.

The Compiler Flags Selection (CFS) tuning is a non-intrusive tuning (i.e. it does not modify the source code). A good knowledge of the compiler intrinsics and of the application is necessary. Although compiler vendors generally propose a set of flags that could improve the execution time, there is no guarantee that these flags are the best possible or even that they will lead to any improvement; for instance, due to the dynamic nature of the application, the sequence of execution may not be clear, or the unrolling factor may be too large and exceed the register capacity.

Table 15 shows the MPI version of NPB running for the three applications with different classes. The compiler flag used to generate these baseline figures was *-O2*. All the runtimes are in seconds.

Benchmarking Scenario		**Number of Processors (nodes x processors)**	
		16 (01 x 16)	144 (04 x 36)
BT-MPI	A	7.05	1.12
	B	92.97	10.32
	C	411.73	44.16
	D	2,461.99	305.88
SP-MPI	A	5.61	2.43
	B	95.09	11.37
	C	544.17	49.14
	D	3,994.9	395.05
LU-MPI	A	4.4	3.55
	B	67.49	18.48
	C	260.13	41.56
	D	3,204.36	192.9

Table 15: Baseline runtime for NPB benchmarks using compiler flag *-O2*

For the CFS manual tuning, we compiled the NPB application with the recommended flags, namely *-O2* (baseline) and *-O3*. Then, we experimented with the selection *-O3 -unroll-aggressive -opt-prefetch* that some benchmarkers proposed for NPB running on Cray systems. Finally, we studied the compiler documentation[25] and composed different sets of flags manually until we found significant improvement with the selection *-O3 -unroll -opt-prefetch -ip -ipo -xhost* for several cases. Omitting *-xhost* led to improvements in certain cases.

Table 16 lists some of the most important optimization flags for the Intel compiler as taken from the Intel Compiler Optimization Guide.

Flag	**Description**
-O2	Maximize speed. Default setting. Enables many optimizations including vectorization.
-O3	Enables -O2 optimizations plus more aggressive loop and memory-access optimizations, such as scalar replacement, loop unrolling, code replication to eliminate branches, loop blocking to allow more efficient use of cache and additional data prefetching.
-parallel	The auto-parallelizer detects simply structured loops that may be safely executed in parallel, and automatically generates multi-threaded code for these loops.
-xhost	Generates instruction sets up to the highest that is supported by the compilation host.
-unroll[n]	Sets the maximum number of times to unroll loops. -unroll0 disables loop

[25] *Quick-Reference Guide to Optimization with Intel® Compilers version 12 For IA-32 processors and Intel® 64 processors (http://software.intel.com/sites/products/collateral/hpc/compilers/compiler_qrg12.pdf)*

	unrolling. The default is -unroll, which uses default heuristics.
-opt-prefetch	Controls the level of software prefetching. n is an optional value between 0 (no prefetching) and 4 (aggressive prefetching), with a default value of 2 when high-level optimization is enabled.
-opt-block-factor=n	Specifies preferred loop blocking factor n, the number of loop iterations in a block, overriding default heuristics. Loop blocking is enabled at –O3 and is designed to increase the reuse of data in cache.
-ip	Single file inter-procedural optimizations including selective inlining within the current source file.
-ipo	Permits inlining and other inter-procedural optimizations among multiple source files. The optional argument n controls the maximum number of link-time compilations (or number of object files) spawned. Default for n is 0 (the compiler chooses).

Table 16: Definition and actions triggered by each Intel compiler flag for code optimization

Table 17 shows the results for different choices of flags for the three NPB applications using two classes of inputs (B and C, which are standard test problems with ~4x size increase from B to C) run on SuperMUC. Table 18 presents the speedup figures for the best performing set of compiler flags found. All times are given in seconds.

Compilation Flags	**BT-SER**		**SP-SER**		**LU-SER**	
	B	C	B	C	B	C
-O2	357.22	1278.87	292.97	1090.46	298.31	1168.09
-O3	353.34	1329.19	313.47	1106.38	309.35	1157.46
-O3 -unroll-aggressive -opt-prefetch	536.09	1299.63	224.98	928.2	218.25	939.63
-O3 -unroll -opt-prefetch -ip -ipo -xhost	331.5	1207.59	250.7	1081.8	208.24	874.34
-O3 -unroll -opt-prefetch -ip -ipo	280.07	1281.82	274.23	1050.88	298.22	1756.07

Table 17: Runtime per compilation flag for NPB benchmarks on SuperMUC

Benchmark		**Best selection of flags**	**Speedup**
BT-SER	B	-O3 -unroll -opt-prefetch -ip -ipo	1.27
	C	-O3 -unroll -opt-prefetch -ip -ipo -xhost	1.06
SP-SER	B	-O3 -unroll-aggressive -opt-prefetch	1.3
	C	-O3 -unroll-aggressive -opt-prefetch	1.17
LU-SER	B	-O3 -unroll -opt-prefetch -ip -ipo -xhost	1.43
	C	-O3 -unroll -opt-prefetch -ip -ipo -xhost	1.34

Table 18: Best performing flags for the NPB benchmarks and speedup figures (“-O2” baseline)

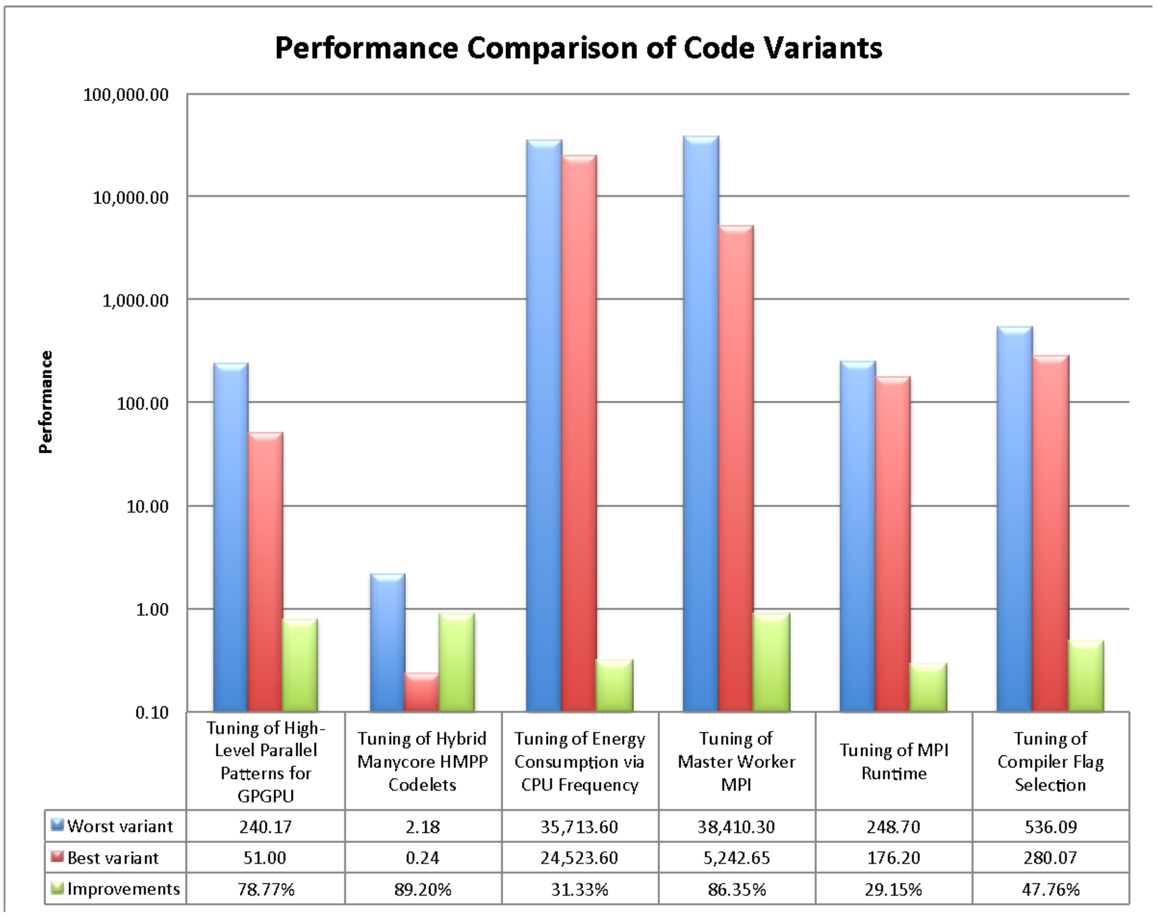
Manually tuning for the best compiler flags is a very time-consuming process. And we found that there is no systematic way to find the best selection. Moreover, finding a selection suitable for a configuration does not guarantee that this selection of flags works on different configurations.

The effort taken for this tuning can be subdivided into the following:

- Learning about the system architecture and the prospective compiler flags for optimizing the application: ~56 man-hours;
- Preparing the tests and writing the test suite scripts on SuperMUC: 8~16 man-hours; and
- Running each test and gathering the results: ~56 man-hours.

4.3 Summary of the Tuning Results

In this Chapter we presented the manual tuning of applications as a tool to hint at the effectiveness and efficiency of tuning techniques. Given each application's performance figures prior to the tuning (baseline values), the developers explored the code variant space using the tuning technique, searching for the best variant in terms of performance. The next figure outlines the performance numbers for the best and worst code variants of the applications optimized per tuning technique. Apart from the results for energy efficiency tuning via CPU frequency (given in Joules), all the other results are given in seconds. Notice the Y-axis is in logarithmic scale.



	Tuning of High-Level Parallel Patterns for GPGPU	Tuning of Hybrid Manycore HMPP Codelets	Tuning of Energy Consumption via CPU Frequency	Tuning of Master Worker MPI	Tuning of MPI Runtime	Tuning of Compiler Flag Selection
🟦 Worst variant	240.17	2.18	35,713.60	38,410.30	248.70	536.09
🟥 Best variant	51.00	0.24	24,523.60	5,242.65	176.20	280.07
🟩 Improvements	78.77%	89.20%	31.33%	86.35%	29.15%	47.76%

Figure 11: Performance for the best and worst variants per manually tuned application

We can see that the techniques were able to find code variants that perform at least 25% better compared to the worst performing variants. Half of the variants excelled in the searching task and were able to find variants 75% better performing than the worst counterparts. All tuning techniques managed to find optimized code variants, in average 60.43% more efficient than the least performing code variant.

With regard to the tuning effort, the following table presents the time taken for the manual tuning. All figures are presented in hours.

Tuning technique	Time effort for manual tuning
High-level parallel patterns for GPGPU	120 man-hours
Hybrid manycore HMPP codelets	140 man-hours
Energy consumption via CPU frequency	112 man-hours
Master-worker MPI	128 man-hours
MPI runtime	120 man-hours
Compiler flag selection	128 man-hours

Table 19: Time effort spent in manual tuning per tuning technique

We can see that all tuning techniques took between 112 and 140 man-hours to be implemented. An average of approximately 125 man-hours of development and tuning time had to be invested in order to properly search the variant space and extract good performance out of the applications.

We believe that the tuning techniques are very promising, given the performance improvements acquired manually compared to the large time effort put onto them. As the tuning plugins will be able to search the code variant space and assess a variant's performance automatically, many more variants can be tested in less time. Thus, better code variants can be found, accounting for higher improvements relative to the worst variants found. All the tuning will be performed without human investment so developer time can be spent in more mentally-challenging tasks.

5 Conclusions

In this document we saw how the Application Repository – the workspace of inputs for test cases – constitutes a central tool in the AutoTune project's evaluation and assessment activities. The repository provides essential feedback to the plugin developers during the design, implementation and integration phases; the test cases not only guide and steer the development of the tuning techniques and plugins, but they are also used to validate them, both in terms of functionality and in performance. We explored the repository's structure, best usage practices and management procedures, which together constitute the foundations for a sturdy evaluation procedure. The features we enforce on the repository are core to a consistent and successful validation process; without them, validating would be too time-consuming, too complex, incorrect, or even just impossible.

Then we introduced the applications we selected for the repository. We described the population where they come from and how the requirements for each plugin have influenced our choice of applications. We introduced the sampling process used to extract the applications and assemble the repository. Finally we depicted all the 9 repository applications one by one, accounting for their features and characteristics, computational kernels, input data sets, and – most importantly – which tuning plugin can exploit it. We demonstrated that the applications chosen for the repository constitutes a solid and cohesive initial set of test cases to steer and validate the tuning plugins and techniques.

To conclude the document, we proved that all 6 tuning techniques are feasible, effective and efficient when manually applied to real-world applications. We described the general tuning procedure common to all techniques, and also the approach taken for each specific technique. In the end, an average of 360 man-hours were invested per application per tuning technique, yielding application performances 60.43% better on average compared to the worst performances. Therefore, since such a laborious manual work resulted in good performance improvements, we conclude that the tuning techniques are indeed promising and worthwhile investing time on, as an automatic search of code variants without human intervention may be able to provide even better performance improvements.

6 Future Works

In this document we saw how the Application Repository – the workspace of inputs for test cases – constitutes a central tool. Although we were able to assemble a repository that covers the plugins' needs well, we understand interesting use cases and scenarios not explored will be discovered. We plan to later expand the repository to account for those applications in order to provide the plugins with a broad overview about the behavior of scientific applications in auto-tuning settings. A future expansion of the repository will also feature more applications that can be optimized by multiple plugins, in order to serve as case studies for the combined plugin tuning strategies.

We especially intend to expand the repository with applications to support the plugin for tuning high-level patterns for GPGPU. We acknowledge the repository can benefit from more pipelined codes and as more applications suit these characteristics the repository should be likewise expanded.

Along with the Periscope and PTF developers at work package 3 "Online Performance Analysis", we also intend to extend the framework to support calling validation functions. The idea is that the plugins are able to assess the conformity of every variant it analyzes, such that code transformations performed by the plugins are guaranteed to yield to a variant equivalent to the original source code.

We will also collaborate with work package 6 "Exploitation and Dissemination" to publicize the results of our manual and automatic tuning efforts. Good improvements figures will be disseminated in scientific and industrial communication vehicles (e.g. workshops, conferences) in due time. Performance improvements will also figure on the website as case studies for PTF. Finally, in case the manual tuning of a community code is performed in a portable manner, the achieved code transformations may be reverted back to the open-source community, benefitting both AutoTune and the community itself.

Appendix A: Specification of the reference machines

Spec Item	SuperMUC	SuperMIG	Nova	Stoney
Login node name	supermuc.lrz.de	supzero.lrz.de	nova0.caps-entreprise.com	stoney.ichec.ie
System model	IBM System x iDataPlex (thin) BladeCenter HX5 (fat)	BladeCenter HX5	Bull Novascale R422-E2	
Processor types (thin + fat)	Sandy Bridge-EP Intel Xeon E5-2680 8C (thin) Westmere-EX Intel Xeon E7-4870 10C (fat)	Westmere-EX Intel Xeon E7-4870 10C	Nehalem-EP Intel Xeon X5560 4C	
Number of islands	18 (thin) + 1 (fat)	1		4 (thin only)
Nodes per island	512 (thin), 205 (fat)	205	20	16
Processors per node	2 (thin), 4 (fat)	4	2	
Cores per processor	8 (thin), 10 (fat)	10	4	
Cores per node	16 (thin), 40 (fat)	40	8	
Logical CPUs per node (HTT)	32 (thin), 80 (fat)	80	8	
Total number of nodes	9216 (thin) + 205 (fat)	205	20	62
Total number of cores	147,456 (thin) + 8,200 (fat)	8,200	160	496
Co-processor types	N/A		NVIDIA Tesla C1060 GPUs	NVIDIA Tesla M2090 GPUs
Co-processors per node	N/A		2	
Peak performance [TFlop/s]	3,185 (thin) + 78 (fat)	78	N/A	37.65 (CPU + GPU)
Linpack Performance [TFlop/s]	2,897 (thin) + 65 (fat)	65	N/A	5.14 (CPU only)
Total size of memory [TiB]	288 (thin) + 54 (fat)	52	0.47	2.91
Memory per core [GiB]	2 (~1.5 thin), 6.4 (~6 fat)	6.4	3	6
Size of shared memory per node [GiB]	32 (thin), 256 (fat)	256	N/A	48
Bandwidth to memory	102.4 (thin),	136.4	30	26

per node [GiB/s]	136.4 (fat)			
Level 3 cache size (shared) [MiB]	20 (thin), 24 (fat)	24	N/A	8
Level 2 cache size [KiB]	256		8,192	256
Level 1 cache size [KiB] and associativity	32 @ 8 way	32	128	32 @ 8 way (data) 32 @ 4 way (instruction)
Level 3 cache line size [B]	64		N/A	8
Expected electrical power consumption of total system [MW]	< 3	< 0.21	N/A	0.03
Network technology	Infiniband FDR10	Infiniband QDR	Infiniband, GigaBit Ethernet	Infiniband DDR
Intra-island topology	non-blocking tree		fat-tree	non-blocking fat-tree
Inter-island topology	pruned tree 4:1	N/A		half-blocking fat-tree
Bisection bandwidth of interconnect [TiB/s]	35.6	N/A		0.0625
File system for SCRATCH and WORK	IBM GPFS	NetApp NAS	Lustre	
File system for HOME	NetApp NAS		NFS	Lustre
Size of parallel storage [TiB]	10,240	N/A	4	21
Size of NAS user storage [PiB]	1.5 (+ 1.5 for replication)	1	N/A (parallel file system only)	
Aggregated bandwidth to/from parallel storage [GiB/s]	200	N/A	25	0.6
Aggregated bandwidth to/from NAS storage [GiB/s]	10	N/A	N/A (parallel file system only)	
Login servers for users	5	2	1	
Service and management servers	12	4	3	1
Batch system	IBM LoadLeveler		SLURM	Torque
Software for archival and backup	IBM TSM		N/A	Bacula
Planned capacity of archival and backup storage [PiB]	> 30		N/A	N/A (only minimal backups)