Project no. 644235

REPHRASE

Research & Innovation Action (RIA)
REFACTORING PARALLEL HETEROGENEOUS RESOURCE-AWARE APPLICATIONS – A SOFTWARE ENGINEERING APPROACH

Requirements on use cases, methodology, and tools
D6.1

Due date of deliverable: 30^{th}, June 2015

Start date of project: April 1^{st}, 2015

Type: Deliverable
WP number: WP6

Responsible institution: Software Competence Center Hagenberg
Editor and editor's address: Michael Rossbory, Software Competence Center Hagenberg

Version 0.1

Project co-funded by the European Commission within the Horizon 2020 Programme		
Dissemination Level		
PU	Public	√
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)	

Change Log

Rev.	Date	Who	Site	What
1	30/06/15	Michael Rossbory	SCCH	Initial version
2	17/10/16	Michael Rossbory	SCCH	Chapter title format, TOC

Executive Summary

This deliverable captures the initial requirements of the RePhrase project, as defined at the start of the project by the project team as a whole, and forming the basis for the successful conduct of the project. It covers technical requirements relating to all aspects of the project as well as requirements on use cases in order to be able to evaluate the whole RePhrase methodology. The deliverable is a key input into all the technical workpackages. This deliverable is intended to be a *living document* since requirements might change over project life time.

Contents

1 Introduction

A first and very important step at the beginning of the project is to define the requirements.

On the one hand the requirements on tools and technologies have to be analyzed that are imposed by the use cases. This is important to ensure that the developed technologies and tools support the needs of developers who aren't experts in programming concurrent software systems.

On the other hand tools and technologies also have demands on the use cases. Defining this requirements ensures that the selected use cases are suitable to evaluate all the tools and technologies and show their benefits in real world applications.

At the beginning of this report the requirements on hardware platform, operating system and compiler are described, followed by the requirements on the tools imposed by the use cases. At the end the demands of the tools on the use cases are specified.

2 Use Cases

2.1 Discrete Optimization

This use case has been derived from a project with a company residing in the area of metal sheet processing. The task is to optimize the production process with regard to multiple parameters like waste of material, production costs or product weight while complying with different constraints. This widespread kind of optimization problem arises in different contexts in various companies. Therefore a generic framework, named *OptiFramework*, has been developed that founds the basis for implementations to solve these kinds of discrete optimization problems.

Based on this framework we implemented an optimization algorithm to optimize the production of transformer cores. The algorithm uses an iterative and stepwise approach and has similarities with the *Simplex Algorithm*. The optimization process consists of two phases. The first phase, that is only executed once at the beginning, an initial admissible solution has to be found that fulfills one required constraint. This partial solution is needed as starting point for the second phase. If such a solution could not be found the algorithm terminates.

Based on the initial partial solution found in the first phase, the second phase iteratively enforces an increasing number of constraints interleaved with optimization steps on the partial solutions. To pursue different optimization paths the initial solution is copied several times and pushed into the pool of intermediate solutions. In each iteration an intermediate solution is picked from the pool and passed through the optimization chain. After an optimization cycle the performance of the solution is inspected. Depending on the new value of the objective function and the optimization history the intermediate solution is considered worth for further optimization and again several copies are pushed into the pool. Otherwise the intermediate solution is dismissed. The optimization process is continued until one of various termination criteria is met, e.g. maximum number of iterations is reached.

2.2 Transfer Learning

Performance of machine learning algorithms heavily depends on the data representation. Building hand-crafted features is often not sufficient to get a proper representation for the supervised machine learning problem at hand [1].

Deep learning has gained increased attention in the last years as a method to learn proper data representations [4]. Deep learning has been inspired by the way the human brain works. E.g. the well-studied visual cortex is hierarchically structured where each level represents increasingly complex and abstract features of the input signal. Deep (neuronal) architectures are therefore composed of levels of non-linear functions allowing them to compactly represent highly non-linear and highly varying functions. These deep learning algorithms try to discover a useful representation at multiple levels, with higher-level learned features defined in terms of lower-level features. These higher-level representations should on the one hand be more abstract and more invariant to the variations in the input distribution and on the other hand preserve as much information form the input as possible [2].

Recently many challenging machine learning tasks have been solved with deep learning approaches, e.g. face recognition [6], image classification [5], toxicity prediction [7]. Actually MIT Technology Review[1] selected it as one of the 10 technological breakthroughs in 2013.

Deep Learning is only enabled with recent availability of very fast computers and massive data sets. Many software tools rely on multi-core and GPU accelerated hardware. Therefore the implementation of novel deep algorithms, e.g. for application specific transfer learning problems [3], requires considerable knowledge of the underlying hardware. This becomes especially relevant if the size of the networks become large and one has manually to deals with the mapping of logical computational units to computation hardware like CPUs and GPUs [5].

A deep learning framework will be developed, which abstracts from the underlying hardware, for representation learning for multi-task learning applications (transfer learning) which are relevant for our industrial partners: deployment of virtual sensors for thousands of different but related scenarios in domains like metal sheet bending and process analytical chemistry. The framework will be used to solve an industrial use case for transfer learning in the area of metal sheet bending. Emphasis will be put on the ability to efficiently develop novel deep architectures and its corresponding learning algorithms. It is intended to exploit and extend the MLPP (machine learning with parallel patterns) framework for this. Methodological contributions to MLPP will be made publicly available in that tool, while data and obtained models have to remain confidential for this use case.

2.3 eRDM: Railway diagnostic system

ERDM is a dynamic railway diagnostic system, which is able to measure the load of each wheel, axle and carriage of the train passing over the system at operating speed. The system also provides diagnostic features; it is able to detect unbalanced rail-cars, wheel flat spots, damaged boogies and suspension problems. Hierarchy of the system can be seen on Figure 2.1.

[1] www.technologyreview.com/lists/breakthrough-technologies/2013/

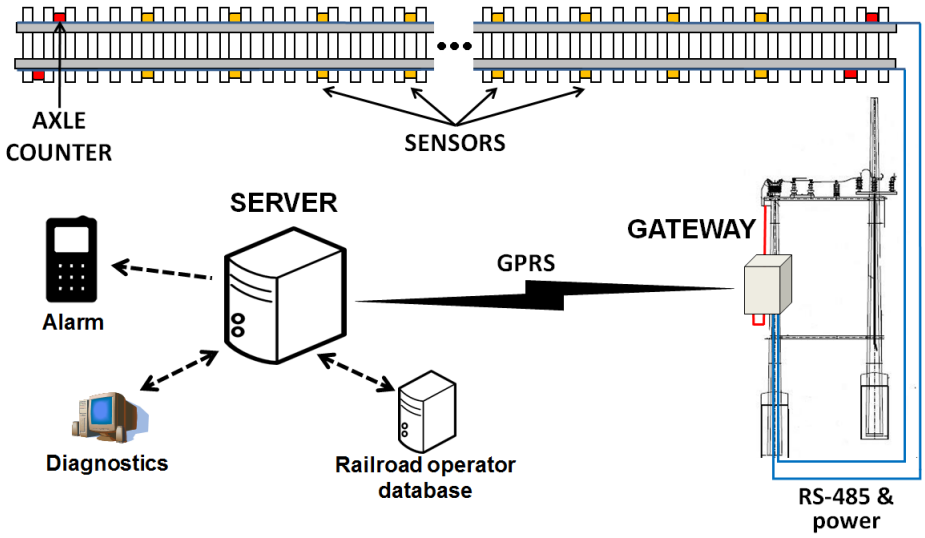

Figure 2.1: ERDM system hierarchy

The system consists of sensor nodes mounted on the rails, a central gateway module processing and forwarding measurement data, a server storing results and different client applications for evaluating measurements. The system is able to send alarm messages real-time if overloaded or dangerous rail-cars were detected. There are 24 sensor modules mounted on the rails pairwise, and four axle counter modules at the two ends of the corresponding rail section. The sensor modules consist of a strain-gauge bridge welded on the rail and a high performance DSP unit driving the bridge and processing its output. The axle counter modules detect if a train reaches the system, initiate the measurement and provide reliable reference data about the position of the train wheels. The gateway module controls the sensor nodes, collects measurement data from them and performs further processing for determining the actual wheel load values.

Both the sensor modules and the gateway unit perform advanced signal processing algorithms, which are to be used as a use case for the Rephrase project. Although this computation is distributed among the sensor modules and the gateway in the current system, they can be executed on a single processing board that directly interacts with the analog hardware. The eRDM use case used in the Rephrase project is a single embedded application integrating both the signal processing algorithm of the sensor nodes and that of the gateway.

2.3.1 Algorithm description

This use case application includes two basic parts: the sensor algorithm performed for each of the 24 analog sensors and the gateway algorithm processing measurement data of the 24 sensors together to produce the load results. The following subsections describes these parts in details.

2.3.1.1 The sensor algorithm

The sensor algorithm includes a signal generator providing excitation signal for the strain gauge bridge. Bridge output is sampled and processed by two parallel FIR (finite impulse response) filters, each consisting of 52 TAPS (filter coefficients). FIR filters are discrete-time linear systems whose current output signal sample can be calculated by convolving the current and previous input signal samples with the filter coefficients. Corresponding bridge amplitude sample is calculated based on the outputs of these filters. There is a separate algorithm implementing a digital circuit, which generates a compensatory signal for the strain gauge bridge. This is responsible for auto-zeroing the bridge. The analog interfaces of the sensor algorithm and the algorithm itself is shown on Figure 2.2.

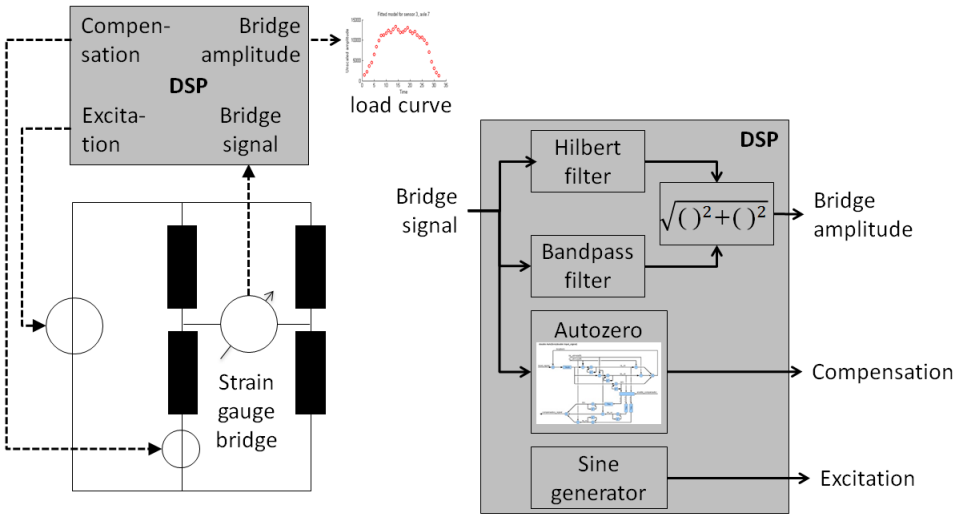

Figure 2.2: ERDM sensor algorithm

If an actual wheel rolls over a sensor, a hill-shaped load curve is generated consisting of samples of the bridge amplitude, whose height is proportional to the actual load. This load curve is produced by detecting when the bridge amplitude exceeds a pre-defined trigger level. The load curve always consists of 64 samples due to a decimation step.

2.3.1.2 The gateway algorithm

If a train with N axles passes over a system, each sensor node produces a load curve of 64 samples for every axle among ideal circumstances. Measured curves corresponding to sensors on the left and right rails are treated separately; thus, samples are arranged in two Nx12x64 arrays. In the next step, height values of the load curves are determined resulting in two Nx12 arrays storing the 12 load values for each wheel of the train. Final load values are then calculated by averaging. This process can be seen on Figure 2.3.

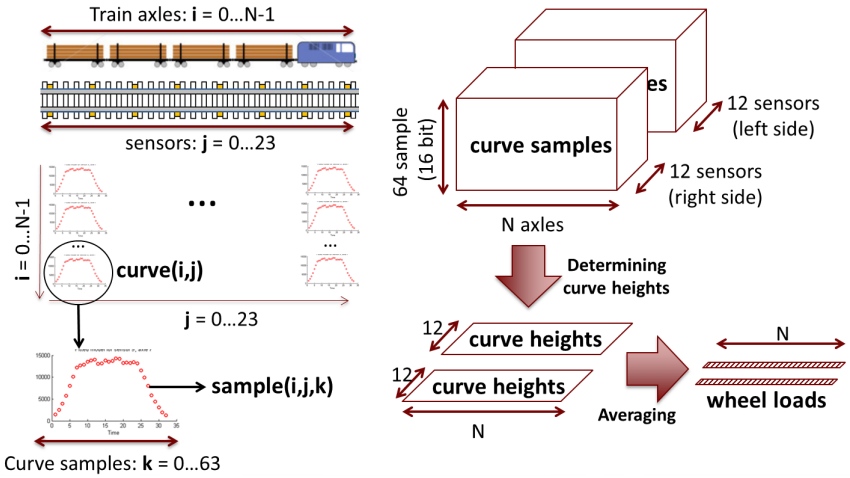

Figure 2.3: Structure of measured and processed data

Figure 2.4 shows the applied algorithm in more details. As it was mentioned, each sensor produces a single load curve for every wheel optimally. However, sensors may miss wheels due to physical conditions; electrical noise can lead to noisy, inappropriate load curves, or even measurement entries which do not correspond to any of the wheels. Thus, the input of the algorithm consists of a set of unaligned curves which are treated initially as unreliable (INPUT box on Figure 2.4). Next, a timestamp alignment algorithm is performed based on the timestamps of the measurement entries. Reliable reference timestamps are provided by the axle counter modules whose analog measurement hardware is more resilient against electrical noise. As a result, the load curves will be assigned to actual axles of the train, and curves that do not correspond to real wheels are discarded (TIMESTAMP ALIGNMENT box on Figure 2.4).

In the next step the heights of the load curves are determined by fitting an ideal curve to them. This is performed by defining and optimizing an objective function with an iterative local search method. The objective function is the root mean square error (RMSE) between the samples of the measured curve and the ideal one; parameters of the function (degrees of freedom of the optimization problem) are a scaling factor and two other values describing the horizontal position of the edges of the ideal curve. When the curve fitting is finished, certain parameters of the measured load curve is determined (using also result of the curve fitting); unreliable and noisy signals are discarded and excluded when calculating final load values (CURVE FITTING & EVALUATION box on Figure 2.4). Final wheel load values are then determined by averaging the load values corresponding to the same wheel (AVERAGING and RESULT boxes on Figure Figure 2.4).

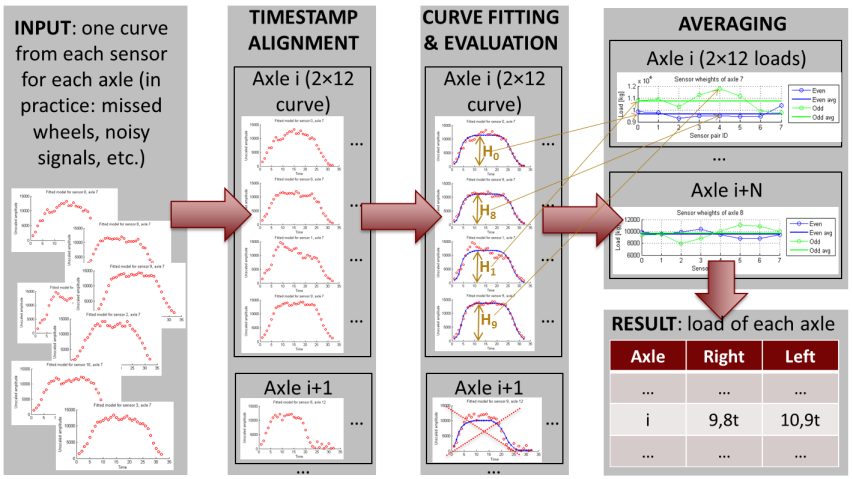

Figure 2.4: The gateway algorithm

2.3.1.3 Use case inputs and outputs

The eRDM use case application used in the Rephrase project processes input files containing simulated analog signals (output of the strain gauge bridges). The application also requires a configuration file describing certain system parameters. Measured load values are written to a textual output file.

2.3.2 Parallelization possibilities

Due to the distributed nature of the original system, the structure of the intermediate data and the applied signal processing algorithms, both the sensor and the gateway algorithm offer various parallelization possibilities, which may be exploited on parallel platforms. Parallelizing the sensor algorithm is possible as follows:

- Since there are 24 sensor nodes in the original system, the sensor algorithm must be executed for 24 different input signal, which can be executed simultaneously by independent processing elements (just like in case of the actual eRDM system).
- The sensor algorithm itself calculates the output of two filters, which can be executed in parallel. Excitation signal generation is also an independent operation, although it is not computationally expensive.
- Calculating the output of the 52-TAP filters requires a convolution operation, which is easily parallelizable; multiplications of different signal sample - filter coefficient pairs as well as summing these products can be executed simultaneously.

The gateway algorithm processes the output signals (load curves) of the sensor nodes, which also includes parallelizable steps:

- The different load curves can be processed in parallel; curve fitting and evaluation, which is the computationally most expensive part of the gateway algorithm, can be performed for every load curve of each sensor node simultaneously.

- Calculating the RMSE value between the measured samples and the currently fitted curve during curve fitting is parallelizable at a low level, since the square of error for different sample pairs can be calculated independently.

- The timestamp alignment algorithm also includes parallel parts, such as the reference timestamp calculation or the alignment of timestamps for independent sensor nodes.

- Calculating the final load values for different axles of the train by averaging can be parallelized, that is, axles can be processed independently, although the required computational effort is low.

As it can be seen, large part of both the sensor and the gateway algorithm is parallelizable; however, the total computational cost of the algorithm is not very high. As a consequence, the parallelization possibilities of eRDM should be exploited by processing elements which are tightly coupled to each other in order to avoid that the communication overhead makes the parallel implementation ineffective.

2.4 Medical Image Processing

This use case is about processing of neuro images in the context of the FSL (`http://fsl.fmrib.ox.ac.uk/fsl/fslwiki/`) open source package. Specifically we will apply the methodology to neuroimage processing at two levels: tractography algorithms (like FSL's *probtrackx*) and intra-voxel reconstruction algorithms (like FSL's *bedpostx*). For these we will apply a reengineering approach which will be representative of software legacy systems adaptation. Algorithms currently developed in MATLAB and will be used as a specification for developing a general and accelerated neuroimage framework.

2.4.1 ProbTrackX

The directional anisotropic information of the diffusion tensor provides a unique opportunity for estimating and modeling the trajectories of white matter tracts in the human brain noninvasively. Algorithms for mapping these connection patterns are referred to as white matter tractography. Deterministic tractography methods are primarily based upon streamline algorithms where the local tract direction is defined by the major eigenvector of the diffusion tensor.

A significant limitation of deterministic tractography methods is the lack of information they provide regarding the error in the tracking procedure in any given

experiment. Without this knowledge it is not possible to know how much confidence we should have in the observed results. Probabilistic tractography methods attempt to overcome this limitation by explicit characterization of the confidence with which connections may be established through the diffusion MRI data set. ProbtrackX is a well-studied method for probabilistic fiber tracking which has been used by several other authors.

Briefly, ProbtrackX repetitively samples from the distributions on voxel-wise principal diffusion directions, each time computing a streamline through these local samples to generate a probabilistic streamline or a sample from the distribution on the location of the true streamline. By taking many such samples FDT is able to build up the posterior distribution on the streamline location or the connectivity distribution. The local diffusion directions are calculated using bedpostx, which allows modelling multiple fibre orientations per voxel. In this Use-Case the sampling procedure can be easily parallelized.

2.4.2 BEDPOSTX

Bedpostx stands for Bayesian Estimation of Diffusion Parameters Obtained using Sampling Techniques. The X stands for modelling Crossing Fibres. Bedpostx runs Markov Chain Monte Carlo sampling to build up distributions on diffusion parameters at each voxel. It creates all the files necessary for running probabilistic tractography. Bedpostx allows to model crossing fibres within each voxel of the brain. For details on the model used in this case, see Behrens et al, NeuroImage 2007. Bedpostx takes about 15 hours to run but will automatically batch if run on an SGE-capable system.

Note that Bedpostx is a wrapper script for a command-line tool called xfibres. This is a very slow process, so bedpost is very processor hungry (a typical Bedpostx run might take around 20hrs for 60 direction 2.5mm isotropic data). However, Bedpostx processes every voxel independently, so it is very easy to parallelise.

3 Requirements on use case, methodology, and tools

3.1 Requirements on Hardware Platform, Operating System and Compiler

The first part of the requirements deals with hardware platform, operating system and compiler issues. It is separated between requirements on the development platform and requirements on the deployment platform. For the deployment platform it is not absolutely necessary that tools that are only needed for development, e.g. refactoring tool or pattern discovery tool, are available there. It is sufficient that all runtime tools, e.g. JIT or dynamic scheduling, are available. For the development platform all the developed tools and runtime tools should be available. Most use cases are more flexible in their requirements on the development platform (e.g. development can be be done either on Linux or Windows), but deployment is often restricted to a certain hardware or operating system (e.g. development is possible using x86 architecture but deployment is restricted to ARM). The following lists the requirements of the use cases on the development and deployment platform:

- **Discrete optimization:** This use case requires a *x86* platform running *Windows* as operating system for deployment. Development is also done on this platform. Tools that only perform static code analysis must at least be available for x86 running Linux. But in terms of software development and user acceptance a developer shouldn't be forced to switch between different platforms during use case development.

- **Deep Learning:** The deep learning framework that will be developed is going to be integrated into the MLPP. The MLPP is intended to be platform independent and available for Linux and Windows running on x86 architecture. Support of *GPU* as accelerator to achieve additional speedup is required.

- **Railway Diagnosis:** This use case will be deployed on an *ARM* platform running *Linux*. The used compiler is Linaro 14.04, an *gcc* 4.8 based compiler. The off-line version used in Rephrase could be executed also on *x86*

with *Linux (Windows)*, but it is preferred to use technologies which are available in the original environment.

- **Neuro-Imaging:** The neuro-imaging use case requires*Linux* on *x86* for both development and deployment. Support of *GPU* is also required.

- **Further considerations:** Strong requirements on this are imposed by the JIT compiler. To evaluate this technology, use cases have to run on *OpenPower* under *Ubuntu Linux* using *LLVM* as compiler. Furthermore the developed patterns have to use *OpenMP*. PRL demands *Solaris* as operating system as another requirement, since it is still widely used by their customers. The refactoring and program shaping tools laid down *Linux* or *Windows* running on *x86* whereat *Linux* on *OpenPower* is also possible. All use cases will be developed in *C++* (at least C++11).

The requirements can be summarized as follows:

- **Hardware platform:** x86, OpenPower, ARM (deployment)
- **Operating System:** Linux, Windows, Solaris9 (deployment)
- **Processing Units:** CPU, GPU
- **Compiler:** gcc, clang, Visual Studio
- **Programming Language:** C++11

3.2 Requirements on Tools

To evaluate the technologies that will be developed in the RePhrase project use cases from different areas, Optimization, Machine Learning, Fault Detection and Medical Image Processing, have been selected. In order to assure that the developed technologies meet the needs of software developers, who are experts in the areas of these use cases but not in parallel programming, the following requirements on tools and technologies have been derived from these use cases.

The following lists requirements that are common to all use cases:

- The developers of the use cases are experts in the area of the use case, but not in parallel programming. Therefore the parallelization process should be guided and automatized as much as possible. Low level details, like locking or mapping, should be hidden from the developer.

- The program shaping tool should support the developer in eliminating race conditions, data dependencies, etc. to prepare the program for parallelization. Program shaping examples are extracting functionality, partitioning data types, removing globals, etc.

- The pattern discovery tool should be able to find code parts that can be parallelized and give suggestions which patterns can be applied e.g. detection of independent loops. If possible the tool should provide performance predictions.
- The refactoring tool should help the developer to introduce patterns that have previously discovered with the pattern discovery tool. At a first step the patterns should be introduced as abstract DSL. Instantiating those patterns into C++ using one of the available technologies (OpenMP, FastFlow, ...) should be separated. Manual refactoring should be reduced to a minimum.
- RePhrase technologies should help the developer in developing parallel applications from scratch to reduce shaping and refactoring afterwards to a minimum.
- All tools that make any changes to the source code have to provide an option for undo/redo, and if possible a preview. Comments should be preserved during code transformation.
- Furthermore code changing tools have to assure that the functionality of the program is left unchanged and that the results are the same before and after each step. Tool support for testing and verifying this is appreciated.
- The use of RePhrase technologies shouldn't force the developer to switch between hardware platforms or operating systems.
- To raise user acceptance the tools should be integrated into commonly used development tools, like Eclipse.
- Very important is to assure that the program leads to the same result before and after parallelization. Some use cases use heuristics based on random number generators
- As stated above some use cases rely on random number generators. RePhrase technologies should provide support for uncorrelated random numbers in multithreaded environments.
- Use cases might work on data that is bigger than the available main memory. An efficient data management strategy has to be provided.
- The use cases are making use of third party libraries like boost, armadillo, opencv, etc. Usage of this libraries has to be possible and should not interfere with any RePhrase technologies.
- As extra functional properties at least energy usage, resource usage (disks, etc.) and performance (throughput, latency, ...) have to be supported.
- Maintainability, testability, etc. have to be preserved.

Additionally to these requirements that the most use cases have in common there a others that are more specific to individual use cases:

Discrete Optimization

- The program consists of two layers. An underlying generic base layer used to solve different optimization problems and a problem specific layer above. The parallelization of the program has to be restricted to the base layer and should not be visible in the layer above.
- Since the algorithm uses heuristics to find an optimal solution, the behavior of the program is not exactly predictable. Work load may change during execution which makes dynamic rescheduling a requirement to achieve optimal performance.
- This use case will be deployed on a x86 system running Windows. The preferred compiler is Visual Studio 13. Therefore a pattern implementation that is fully compliant with this compiler is needed (e.g. OpenMP could be problem since the latest standard is not supported).
- The more partial solutions can be processed in a certain time frame the more likely it is to find an optimal solution. Therefore the primary metric to evaluate the performance is the number of processed partial solutions per time frame. Extra functional properties like power consumption have less significance.
- The initial sequential version has been developed without having parallelization in mind. During further development some functionality has been added, some removed. As a consequence the architecture got complicated with dependencies and connections that are hard to detect and to eliminate. A sophisticated support for shaping and refactoring is highly demanded.

Machine Learning (DeepLearning)

- This use case will be developed from scratch. To minimize shaping and refactoring effort afterwards RePhrase technologies should guide the developer from the beginning.
- Training deep architecture, especially when architecture selection is performed is computationally very intensive. Therefore support of accelerators (GPU) is required.

eRDM (railway diagnosis)

- As a use case we use a version of the software which processes pre-captured measurement data offline, but it should be able to process input data at least with the throughput as if the data would be captured real-time (4 byte samples with 62,5kHz frequency from 24 sensors = 6MByte/sec).

Neuro-Imaging

- Hardware architecture: both Intel x86 and Power8 (OpenPower).
- Support for both Linux and Windows operating systems in order to extend portability to final users.
- Integration of tools into Visual Studio.
- Programming language: GCC 5.0 and LLVM (CLang) under GPUs.
- Accelerator runtime: CUDA (OpenCL as second option)
- OpenMP in order to provide support for multiple GPU deployment.
- The usage of smart pointers to optimize performance in presence of large data sets.

3.3 Requirements on Use Cases

All the technologies developed in the RePhrase project should be evaluated using real world applications. Following requirements on use cases have been identified to assure that all the tools and technologies can be evaluated.

- A fully-compliant C++14 version of the legacy code that supports parallel patterns.
- An optimized sequential version of the software in order to verify that the results are obtained correctly is needed.
- The results of the use case have to be reproducible.
- It is important to check out that the obtained results are similar independently of the architecture.
- In addition, a strong requirement is that the code to be processed should be parallelized only using patterns (no additional, extra-pattern parallelism), or with patterns and additional parallelism expressed using frameworks/tools that do not interfere with the pattern implementation (to be identified sometime early in the project).
- Use of template meta programming or over-complicated macros should be avoided.
- To test certain behavior of the tools (e.g. GPU offloading under certain conditions) synthetic data for the use cases should be available. Otherwise some synthetic benchmarks have to be developed. But the use cases should not be artificially hacked to force certain runtime behavior.

- To test tool interoperability at least for one use case development and deployment should be possible on different hardware platforms and operating systems.
- Use cases should be flexible enough to allow for parameter tuning (i.e. should not strictly fix granularity, work division, work placement etc.)
- When using the pattern library the use case developers should not explicitly see which technology has been used for pattern implementation. For comparison of manually implemented code vs. developed using RePhrase technologies concerning evaluation metrics that are going to be defined the use cases might have to be also implemented using technologies like OpenMP or CUDA.
- Pattern implementations need to expose modifiable extra-functional parameters as an interface.
- Extra-functional parameters: number of cpu-gpu workers in farms, granularity/chunking parameters, data-placement parameters, work division (between CPUs and GPUs)

4 Conclusion

This initial collection of requirements is based on properties of use cases described at the beginning, experiences in software development, in particular of concurrent software systems, and experiences made in former projects. During the project additional requirements will arise, which we are not aware of at the beginning of the project. Furthermore it is possible it turns out that some of the requirements defined in this document are not realizable within the duration of this project. Therefore the content of this deliverable might change during project time.

Bibliography

[1] Yoshua Bengio. Learning deep architectures for AI. *Foundations and TrendsÂő in Machine Learning*, 2(1):1–127, 2009.

[2] Yoshua Bengio. Deep learning of representations for unsupervised and transfer learning. In *JMLR: Workshop and Conference Proceedings*, volume 7, pages 1–20, 2011.

[3] Yoshua Bengio. Deep learning of representations for unsupervised and transfer learning. *Unsupervised and Transfer Learning Challenges in Machine Learning, Volume 7*, page 19, 2012.

[4] Yoshua Bengio and Aaron Courville. Deep learning of representations. In *Handbook on Neural Information Processing*. Springer, 2013.

[5] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[6] Yi Sun, Xiaogang Wang, and Xiaoou Tang. Deep learning face representation from predicting 10,000 classes. In *Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on*, pages 1891–1898. IEEE, 2014.

[7] Thomas Unterthiner, Andreas Mayr, GÃijnter Klambauer, and Sepp Hochreiter. Toxicity prediction using deep learning, 2015. arXiv:1503.01445, http://arxiv.org/abs/1502.06464v1.