Project no. 644235

REPHRASE

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

Report on defined Use Cases
D6.3

Due date of deliverable: April 2016

Start date of project: April 1st, 2015

Type: Deliverable
WP number: WP6

Responsible institution: SCCH
Editor and editor's address: Michael Rossbory, SCCH

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	30/04/16	Michael Rossbory	SCCH	Initial version
2	07/10/16	Michael Rossbory	SCCH	Chapter references, toc

Executive Summary

This deliverable D6.3 (Report on Defined Use Cases) describes the use cases selected by the partners (SCCH, evoPro, cibersam) contributing to WP6 (Use Cases, Metrics and Evaluation.

For every use case a detailed description of the use case background and environment is given and the current implementation (if already available) is explained. Furthermore, input andd output data for the use case applications are specified. Where available, the results from the application profiling are outlined, along with parallelization possibilities derived from these results and the algorithm implementation. Additional requirements that are not already captured by by D6.1 (Requirements on use cases, methodology, and tools) are described. Additionally, proposed use of methodologies developed in the RePhrase project is outlined.

All use cases are real projects running at the corresponding partner. Therefore, some of them are subjects of restriction imposed by the customer. Details of the use cases are provided as far as they do not violate any of the confidentiality issues.

Contents

1 Introduction

The main purpose of WP6 is evaluation of the **RePhrase** methodology and tools that will be developed in the technical workpackages (WP2, WP3, WP4, WP5) with respect to their practical applicability in real world applications. Results of this evaluation flows back to the technical workpackages for their improvement.

Furthermore, the use cases will be used for dissemination of the project achievements among the partners and within the domain of the WP6 consortium.

To ensure applicability to a wide range of application domains WP6 partners come from multiple different domains: agile custom made software development for data analysis and computer vision (SCCH), distributed real-time industrial system monitoring (EvoPro), and bio-medical image processing (cibersam).

The following list contains a very brief overview of the selected use cases that will be described in detail in the remaining chapters of this deliverable:

Railway Diagnosis (evopro) eRDM is a dynamic railway diagnostic system, that 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. Thus the system allows real-time freight transportation monitoring, helps increasing railway transportation safety and reducing rail/carriage maintenance costs.

Medical Imaging (cibersam) This use case is about processing of neuro images in the context of the FSL open source package. Specifically we will apply the methodology to neuroimage processing at two levels: tractography algorithms (like FSLs probtrackx) and intra-voxel reconstruction algorithms (like FSLs 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 neuroimage framework.

Stochastic Local Search (SCCH) Effective utilization of all kind of resources, from machines and raw material to energy and human resources, and highly optimized process cycles are crucial for every company. This particular use case deals with an optimization problem encountered in the slitting of

metal sheets used in the production of electrical transformers. The problem is a generalized version of the so-called 1/V/V/R cutting-stock problem, since the objective corresponds to appropriately placing a set of metal stripes (bands) into a set of available metal coils, so that the overall metal waste is minimized. Optimization algorithms like this are computationally very intensive and parallelization is crucial to achieve good results within a reasonable time.

Weather Forecast (SCCH) Systems that are heavily influenced by environmental impacts like temperature, precipitation or solar radiation depend on precise local forecasts to achieve planning reliability. The spatiotemporal resolution of available global weather models is too coarse grained to take geographical characteristics into account, which are essential to make local-optimized weather predictions. In this use case new weather forecast models based on prediction data from global models using different machine learning methods will be developed. The training of these models is computationally very intensive and uses huge amount of historical weather data. Parallelization therefore, is essential in training as well as in model application.

2 Railway Diagnosis

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. Thus the system allows real-time freight transportation monitoring, helps increasing railway transportation safety and reducing rail/carriage maintenance costs. Hierarchy of the system can be seen on Figure 2.1.

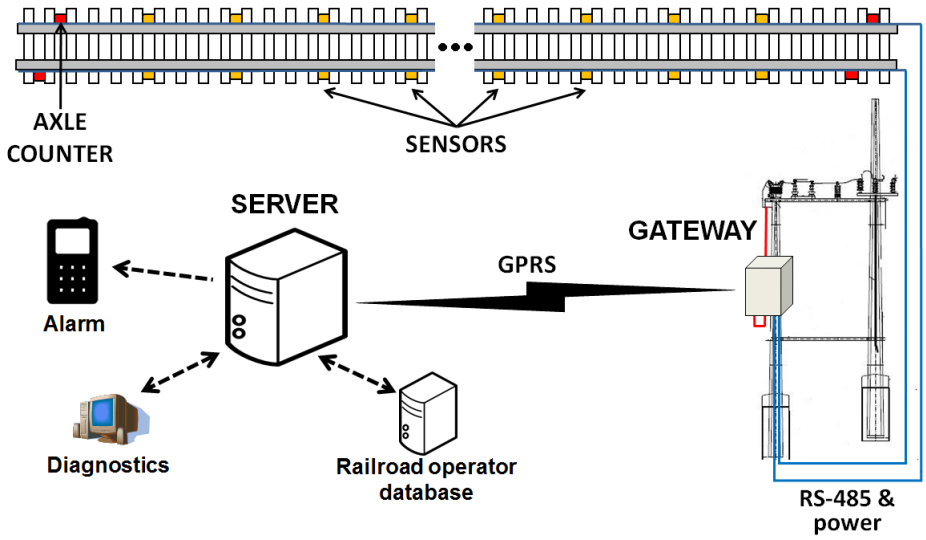

Figure 2.1: ERDM system hierarchy

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. This architecture would be advantageous since the DSP processor units could be omitted from the system reducing its cost; however, the required computational cost of the algorithm running on the gateway would drastically increase. The gateway is a industrial embedded computer. Similarly to the desktop/server/mobile platforms, embedded computers tend to include multiple processor cores and in some cases even dedicated graphical processing units (e. g. the Jetson developer kit from NVIDIA utilizing a Tegra Soc with a quad-core ARM and a Kepler GPU with 192 CUDA Cores). 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. If this application can be effectively parallelized and executed on a heterogeneous computing platform, the approach of applying a single high-performance embedded computer instead of the current distributed computing architecture in the eRDM system is justified.

2.1 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.

2.1.1 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, which implement actually envelop detection. 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.

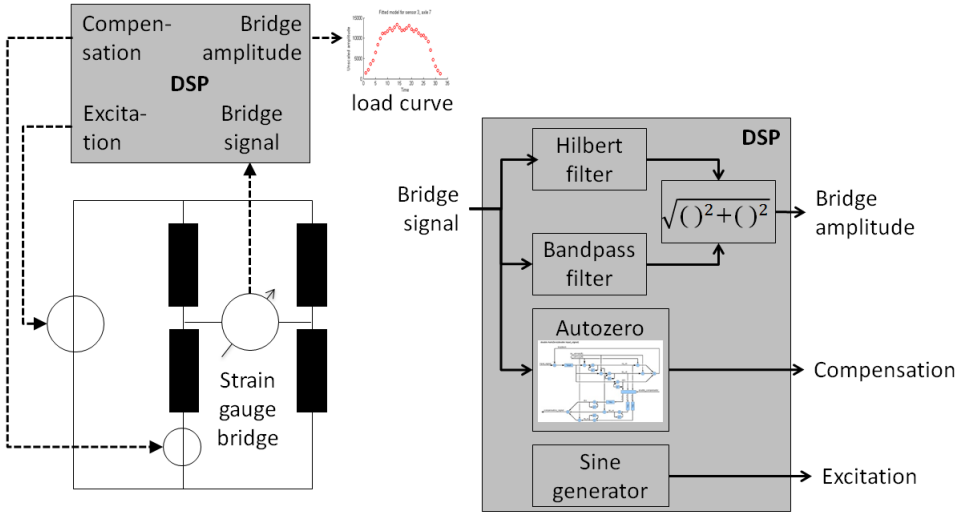

Figure 2.2: ERDM sensor algorithm

If an actual wheel rolls over a sensor, a hill-shaped load curve is generated consisting of samples of the bridge amplitude, which is determined by a triggering algorithm executed on the filtered (envelope) signal. Height of a load curve is proportional to the actual load. The load curve always consists of 64 samples due to a decimation step.

2.1.2 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 2.4 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 ALIGN-

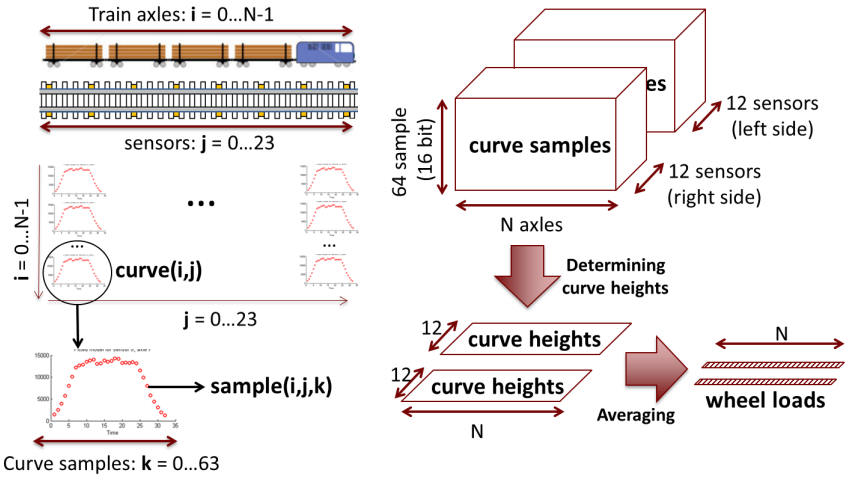

Figure 2.3: Structure of measured and processed data

MENT 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 2.4).

2.1.3 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 and diagnostic values are written to a textual output file.

2.2 Profiling

Figure 2.5 outlines the eRDM agorithm code. It is not an extract from the real code, but shows the basic structure of the algorithm, highlighting the parallelization possibilities. Measured signal is processed for each sensor (*Loop 1*). Within this loop, each output sample is calculated in *Loop 2* as a convolution of N input

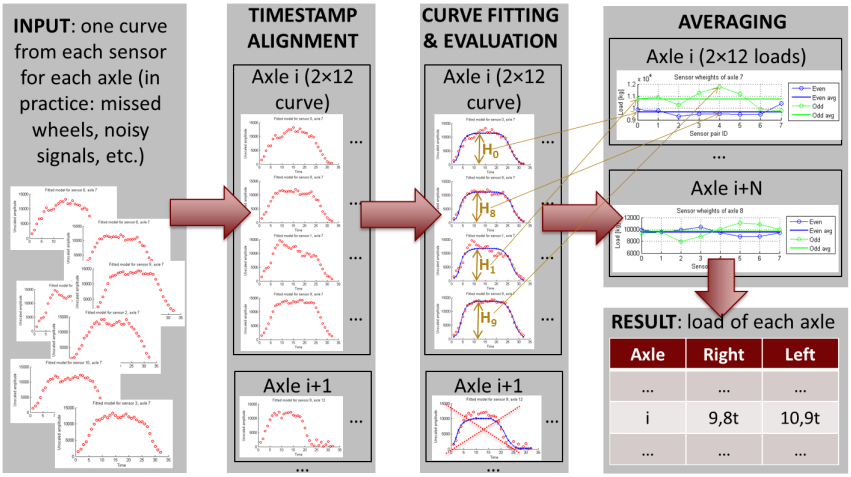

Figure 2.4: The gateway algorithm

samples (performed actually twice with different filter coefficients within the function *calcFilterOutput()*, just as shown on Figure 2.2). After filtering, load curves are extracted from each envelope signal by the *trigger()* function, then timestamp alignment is performed. In *Loop 3* each load curve is processed (curve fitting and evaluation, see also 2.4). The last step is calculating the results (load, diagnostic parameters) for each axle (*Loop 4*).

The use case application was compiled with -O3 flag and run on a single core of a high-end CPU (Intel Xeon E5-2695 v2). Execution time of the different code parts shown on Figure 2.5 was measured. Table 2.1 summarizes the results, which were acquired as the average of several distinct runs.

Code part	Execution time
calcFilterOutput()	6.4s
trigger()	0.14s
align()	0.002s
fitEvaluateCurve()	0.61s
calcLoadDiag()	0.0001s
file processing	2.3s
other	0.35
total program	9.8s

Table 2.1: Average run times of eRDM code parts

The total program run time was 9.8s, out of which 2.3s was spent on file processing and other housekeeping tasks which occur only in the offline version of the application. Later, if is executed in a real environment, this part would be omitted.

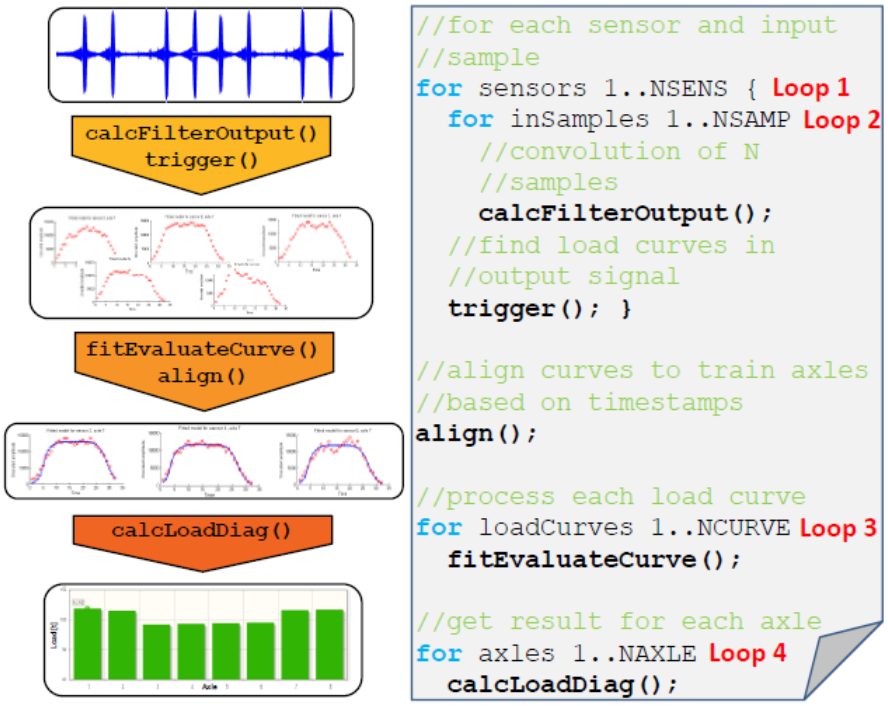

Figure 2.5: Overview of eRDM algorithm code

Another important point is that although the total execution time seems to be quite low, if the algorithm is executed on an embedded architecture, which represents much lower computational power than a server CPU, the execution time would increase. Also, the application was run supposing 24 sensors and a 104 axle train with 60km/h speed. Number of sensors may be higher in an actual system for the sake of redundancy and robustness. Number of axles and speed of train are arbitrary values during a real measurement. Both of these factors (number of sensors, measurement length) influence the amount of input data, and thus the execution time linearly (or almost linearly). However, in a real system the execution time must be as low as possible so that measured signals can be processed as fast as they are acquired, and alert conditions (overloading, damaged wheel, etc.) can be recognized and sent real-time which allows the train to be stopped when it reaches the next station. So that these requirements could be satisfied even on an embedded computing architecture, parallelization is necessary.

As it can be seen in Table 2.1, the largest part of the actual computation time (6.4s) was spent in the *calcFilterOutput()* function, that is, in *Loop 2* performing filtering. *Loop 1* also includes the *trigger()* function. Both *Loop 1* and *Loop 2* are hot-spots in the application; major speedup can only be achieved if these loops are parallelized. *Loop 3*, that is, curve fitting and evaluation accounts for 0.61s, which is not negligible compared to the total run time. Also, the curve evaluation algorithm might be improved later to enhance diagnostic functionality (e. g. accuracy

of wheel flat spot measurement), which can increase the amount of computations required. All the other code parts represent only minor computation; optimizing their execution speed would virtually have no effect on the total program run time.

2.3 Parallelization possibilities

The eRDM algorithm offers several ways of parallelization; it is worth considering how they are related to the typical parallel patterns such as pipeline, taskfarm, map-reduce, etc. All the four loops shown on Figure 2.5 consists of independent iterations, which could be transformed to a taskfarm or map (stencil) pattern. *Loop 1* represents slightly higher computation than *Loop 2*. The main difference between them is that the former iterates over the sensors, so the number of parallel work items (possibly workers/threads) would be equal to the number of sensors (typically 24). In case of *Loop 2* the iteration is performed over the input/output samples, thus the amount of parallel work items may be in the range of 10^5-10^7. The former approach suits a processing unit with lower number of high-performance cores (i. e. a multicore CPU), the latter one a device with a lot of cores (e. g. a GPU). Another possible way of parallelization is executing the *caclFilterOutput()* function itself in parallel, which essentially implements a filtering/convolution operation for 52 sample-coefficient pairs (multiply-accumulate operations), repeated for two sets of coefficients. This is inherently a map-reduce pattern, although the parallel operations are in this case simple multiplications (map) and accumulations (reduce).

Loop 3 also consists of independent iterations and could be transformed to a taskfarm/map pattern. The number of load curves (number of parallel work items) are usually equal to the number of sensors multiplied by the number of axles (each sensor measures each wheel represented by a load curve), so it is in the range 10^2-10^3. The *fitEvaluateCurve()* function inside (not shown on Figure 2.5) consists of a couple of sequential steps (curve fitting, curve evaluation), which could be exploited by a pipeline pattern.

As it was mentioned earlier, *Loop 4* accounts for only negligible part of the program run time, accelerating it is not feasible.

2.4 Requirements

In conclusion, applying the typical parallel patterns (most notably taskfarm and map/stencil) for parallelizing the eRDM algorithm seems to be very promising. It is essential that the RePhrase tools support these patterns in order to generate parallel versions of the eRDM application easily. Technologies (parallel libraries, etc.) which are available on ARM CPUs are preferred, since most likely the algorithm will run on such an architecture eventually.

Another important point is that the application (namely, the *fitEvaluateCurve()* function performing curve fitting with heuristic algorithm) uses random numbers,

so a thread-safe parallel random number library is required.

Since in a real environment the application would process sensor signals on-the-fly, it is important that the speed of signal processing outperforms that of the data acquisition, which is cca. 6MByte/sec for 24 sensors. Considering the offline version of the application, the total execution time should be below the theoretical data acquisition time (which is probably easily satisfied on a high-end server CPU, but would be challenging on an embedded CPU).

3 Medical Imaging

3.1 Algorithms description

After decades of developments in Magnetic Resonance Imaging (MRI), the successful implementation of a variety of advanced methods and image modalities have shed light on the complex patterns of brain organization present at micro and macroscopic scales. The structural organization of the white matter of the brain can be depicted in great detail with advanced diffusion weighted magnetic resonance imaging (DW-MRI) schemes. DW-MRI offers, through tractography approaches, the only way to study brain structural connectivity, non-invasively and in-vivo.

A straightforward image processing pipeline for Diffusion Weighted Images (DWIs) is shown at Figure 3.1. This pipeline involves four main steps:

1. *Image correction*: Preprocessing steps to reduce noise, increase contrast and remove distortions and movements effects from DWIs.

2. *Intravoxel fiber orientation reconstruction*: This step allows for estimating the tridimensional nerve fiber orientation inside each image voxel.

3. *Fibre tracking*: This stage employee the orientational information computed in the previous step to construct digital paths along brain white matter. These paths are a digital and tridimensional representation model of brain white matter tracts.

4. *Anatomical connectivity computation*: The last step of these kind of image processing pipelines is to compute the connectivity between brain gray matter regions.

The CIBERSAM "use case" is focused on the intravoxel fiber reconstruction and fiber tracking steps due to the long computation time, since depending on the parameters of the MRI acquisition protocol, the analysis of a single dataset can easily take more than 24 hours on a single-core CPU.

Our use case comprises the application of the RePhrase approach to neuroimage research in the context of:

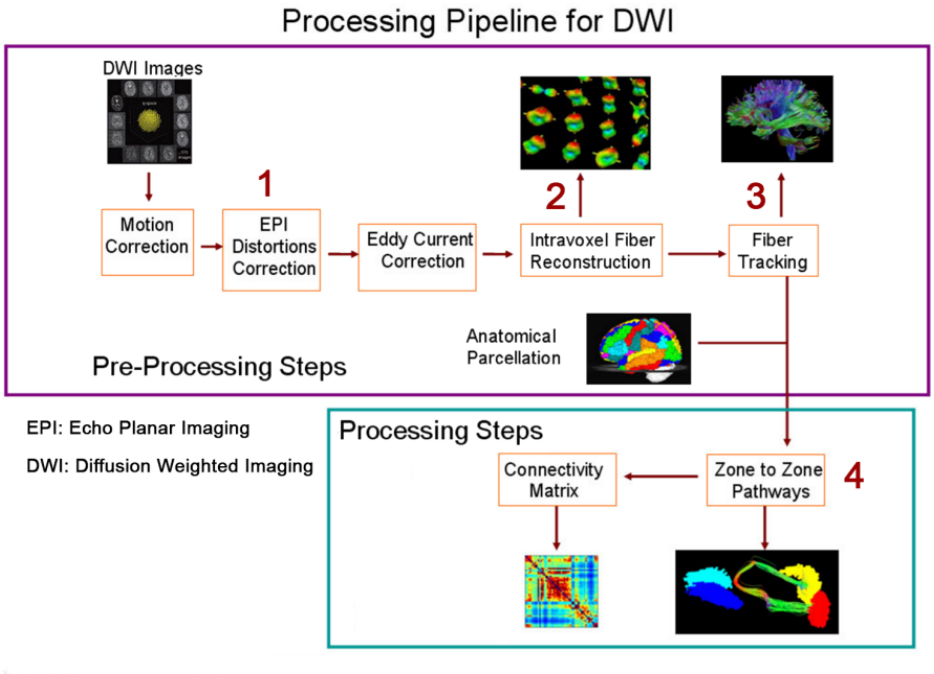

Figure 3.1: Diffusion Weighted Imaging Processing workflow.

- The open source package FSL[1]. Specifically, we will apply the methodology to tractography algorithms (like FSL's probtrackx) and intra-voxel fiber reconstruction algorithms (like FSL's bedpostx). For these, we will use a reengineering approach that will be representative of software legacy systems adaptation.

- Published algorithms currently developed in Matlab at CIBERSAM [2]. For these new algorithms we take the MAtlab source code as detailed requirement specifications. We will apply the full RePhrase methodology from requirements gathering to verification and system deployment. With this approach we will produce a new implementation in C++ where parallelism has been considered throughout the full software development lificycle.

3.1.1 FSL 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 needed files 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 [1]). Bedpostx takes about

[1] http://fsl.fmrib.ox.ac.uk/fsl/fslwiki/

15 hours to run but will automatically batch if run on an Sun Grid Engine (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 directions and 2.5mm isotropic data). However, Bedpostx processes every voxel independently.

The resultant 6D image represents the function of three position variables (3D position vector) and three displacement variables (3D displacement vector) as shown in Figure 3.2.

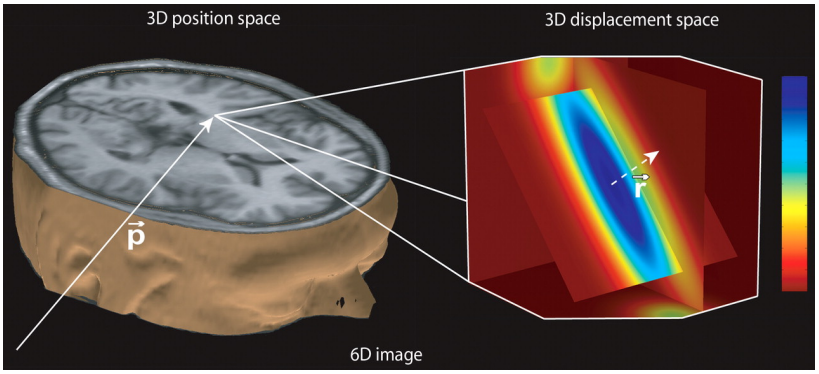

Figure 3.2: Resulting 6D image from Bedpostx.

For each voxel/pixel in a 3D space, the application studies the plane orientation in a 3D space. In Figure 3.2, the zoom of the right represents a water molecule and its orientation in the space, depending of the magnetic emission.

3.1.2 FSL probtrackx

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 (as shown in Figure 3.3). 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 modeling multiple fibre orientations per voxel.

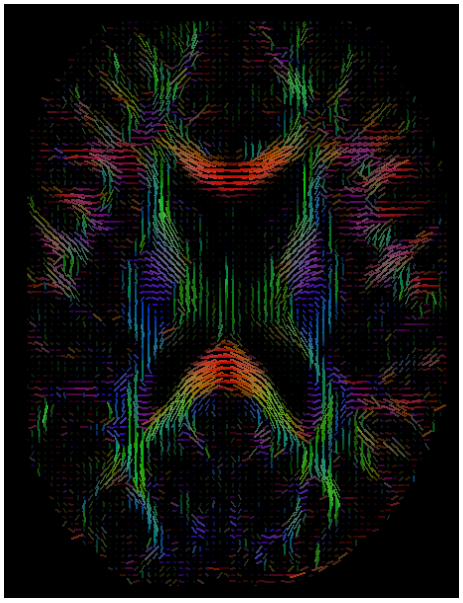

Figure 3.3: Main diffusion representation.

One we have recognized the water molecules, the objective is to locate the brain pathways. The method consists on tracking the pathways given the water molecules orientation. Water molecules are close to fibers. The algorithm finds the most suitable pathway parting from a seed region. The algorithm deals with the seed's neighborhood in a probabilistic way. The most probable pathways have a bigger value. All the pathways found are colored in terms of its value.

3.1.3 Intravoxel Fiber Reconstruction (Rumba)

In biological tissues, water diffusion is not equal in all directions due to the presence of obstacles or barriers which limit the molecular movement. To quantitatively characterize this phenomenon the diffusion tensor imaging (DTI) is used. This tensor is defined as a 3x3 symmetric and positive defined matrix.

$$D = \begin{pmatrix} D_{xx} & D_{xy} & D_{xz} \\ D_{yx} & D_{yy} & D_{yz} \\ D_{zx} & D_{zy} & D_{zz} \end{pmatrix} \tag{3.1}$$

The diffusion tensor may be visualized as an ellipsoid, where its orientations are defined by its eigenvectors and the magnitude of the principal axis are defined by the mean diffusion for a given eigenvalue at a given diffusion time (Figure 3.4).

Through the eigenvalues of the diffusion tensor, it is possible to infer qualitative characteristics of the spatial arrangement of the fibers inside a voxel (Figure 3.4). They can be grouped into three convenient classes: class II describes the case where $\lambda_1\lambda_2 \simeq \lambda_3$. In this case there is a well-defined dominant diffusion direction

(the principal eigenvector, e1, with magnitude l1) and the ellipsoid takes on a cigar-shaped (prolate) form (Figure 3.4a). Class II describes the case where $\lambda_2 \simeq \lambda_3 \lambda_3$ (Figure 3.4b). In this case the dominant direction of diffusion is less well-defined, and the underlying orientation of fibers is ambiguous. In this case the ellipsoid is disk-shaped (oblate). Class III describes the case where $\lambda_1 \simeq \lambda_2 \simeq \lambda_3$. In this case diffusion is isotropic, and the ellipsoid equates to a sphere (Figure 2c). This is the form of tensor expected in free diffusion, and will be observed in cerebrospinal fluid and may closely be approximated in grey matter, where no dominant underlying structural orientation exists.

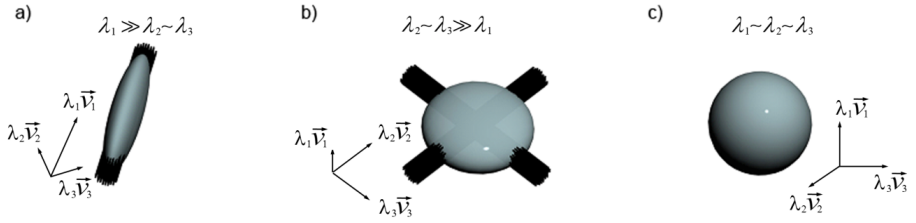

Figure 3.4: Microstructural organization can be revealed by the diffusion tensor ellipsoid.

DTI can deliver quantitative results, it may be easily implemented in any clinical MRI system and, thanks to its short acquisition time, it may be suitable for studying a wide range of brain diseases. Unfortunately, it is now well recognized that due to its simplistic assumptions, the DTI model does not adequately describe diffusion processes in areas of complex tissue organization, like in areas with kissing, branching or crossing fibers (Figure 3.5).

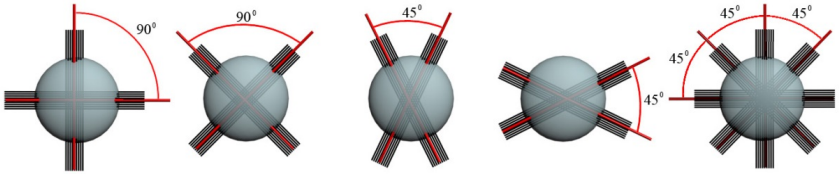

Figure 3.5: Different fiber crossing can give the same diffusion ellipsoid.

Such limitations in the DTI approach have prompted the recent development of numerous sampling protocols, diffusion models and reconstruction techniques

Intravoxel fiber crossing can be resolved by computing the Orientational Distribution Function (ODFs) from DWIs and water displacement distribution. An orientation distribution function may be considered a deformed sphere whose radius in a given direction is proportional to the sum of values of the diffusion probability density function in that direction (Figure 3.6). An orientation distribution function or isosurface can be plotted for each individual MR imaging voxel in a section.

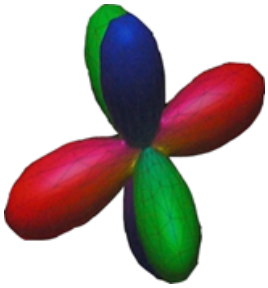

Figure 3.6: Orientational Distribution Function. For ease of visualization, we color code the surface according to the diffusion direction ($[x, y, z] - [r, b, g]$, where r=red represents probability along left-right orientation, b=blue represents inferior-superior orientation, and g=green represents anterior-posterior orientation).

3.1.3.1 Simulated data

The diffusion MRI signal measured for a given voxel can be expressed as the sum of the signals from each intra-voxel compartment. The term "compartment" is defined as a homogeneous region in which the diffusion process possesses identical properties in magnitude and orientation throughout, and which is different to the diffusion processes occurring in other compartments. One example of this approach is the multi-tensor model that allows considering multiple WM parallel-fiber populations within the voxel. In this model the diffusion process taking place inside each compartment of parallel fibers is described by a second-order self-diffusion tensor. This approach can be used to simulate diffusion MR signal throughout the following equation:

$$S(b, g) = S_0 \sum_{j=1}^{N} f_j \cdot exp\left[-bg * D_j * g^T\right] \tag{3.2}$$

Where f is the compartment volume fraction, b is a parameter that depends of the experiment and D_j is the compartment diffusion tensor.

Given a fiber configurations, the diffusion MR signal is simulated in each voxel using the previously shown expression (Figure 3.7 left).

As for the local model of diffusion, the signal is simulated considering hindered and restricted diffusion, to account for **extra-axonal and intra-axonal diffusion**. Finally, depending on the position in space, there is also an isotropic compartment, to account for the **CSF contamination** close to the ventricles in brain imaging.

The magnitude MR signal is corrupted by **Rician noise**. If S is the noise-free diffusion weighted signal, then the signal is simulated as:

$$S_{noisy} = \sqrt{(S + \varepsilon_1)^2 + \varepsilon_2^2} \tag{3.3}$$

Where ε_1 and ε_2 and $N(0, \vartheta)$ and $\vartheta = S_0/SNR$.

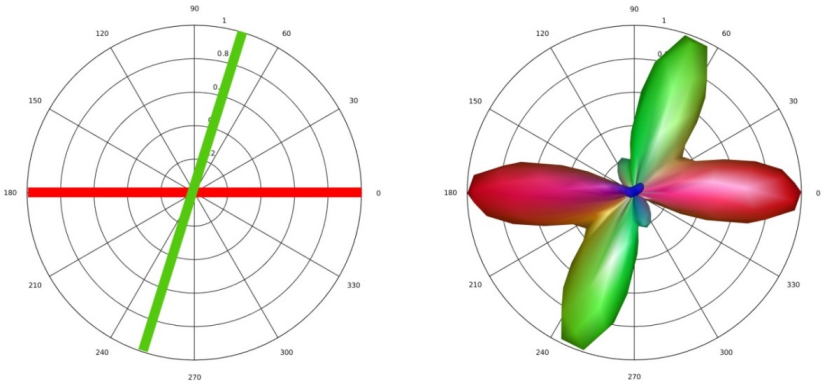

Figure 3.7: Simulation of a fiber crossing between two fibers (Left panel). Reconstructed ODF using RUMBA (Right panel).

The simulated signal for a given SNR and a given acquisition scheme is provided as a **4D Nifti image**.

3.1.3.2 Real data

A subset of 100 unrelated subjects (age range between 26 and 31 years) were selected from the Human Connectome Project (HCP) [8] database[2]; sixty five of them contained also a follow up scan. These subjects were selected due to the high quality of the data, which will allow for an easier implementation of the designed procedures, although the full robustness of the experiments must be reassured.

All subjects were scanned on the same 3T Siemens Skyra platform with a 32 channels head coil at Washington University (WashU). The scanning protocol includes four imaging modalities: structural MRI, resting-state fMRI (rfMRI), task fMRI (tfMRI), and diffusion-weighted MRI (dwMRI) but only the T1W-MRI 3D MPRAGE and the DW-MRI were selected for all the participants. The scanning parameters for both MR sequences (T1 and DW) are shown at Figure 3.8.

In the following figure we can observe the fractional anisotropy map and reconstructed ODFs obtained by RUMBA algorithm for one healthy volunteer.

3.2 Data management

3.2.1 Bedpostx and Probtrackx

Input and Outputs for Bedpostx and Probtrackx (taken from bedpost web page http://fsl.fmrib.ox.ac.uk/fsl/fslwiki/FDT/UserGuide).

[2] http://humanconnectome.org/documentation/data-release/Q1_Release_Appendix_I.pdf

Number of Subjects: 100 unrelated subjects
Age range: 26 – 31 years

Siemens Skyra (3T) platform with 100 mT/m gradients for diffusion encoding and ~ 42 mT/m gradients for imaging.

Structural Images

Type	T1w	T2w
Description	3D MPRAGE	3D T2-SPACE
TR (ms)	2400	3200
TE (ms)	2.14	565
TI (ms)	1000	
Flip Angle	8 deg	variable
FOV (mm)	224x224	224x224
Voxel Size	0.7 mm isometric	0.7 mm isometric
BW (Hz/Px)	210	744

Diffusion Weighted Images

Parameter	Value
Sequence	Spin-echo EPI
TR	5520 ms
TE	89.5 ms
flip angle	78 deg
refocusing flip angle	160 deg
FOV	210x180 (RO x PE)
matrix	168x144 (RO x PE)
slice thickness	1.25 mm, 111 slices, 1.25 mm isometric voxels
Multiband factor	3
Echo spacing	0.78 ms
BW	1488 Hz/Px
Phase partial Fourier	6/8
b-values	1000, 2000, and 3000 s/mm2

Each gradient table includes approximately 90 diffusion weighting directions plus 6 b=0 acquisitions interspersed throughout each run

Figure 3.8: Data and image acquisition specifications for the images used in this Project.

3.2.1.1 Inputs

- data: A 4D series of data volumes in Nifti image format. This will include diffusion-weighted volumes and volume(s) with no diffusion weighting.
- nodif_brain_mask: 3D binary brain mask volume in Nifti image format.
- bvecs (with no file extension): An ASCII text file containing a list of gradient directions applied during diffusion weighted volumes. The order of entries in this file must match the order of volumes in data. The format is:

$$\begin{matrix} x_1 & x_2 & x_3 & \dots & x_n \\ y_1 & y_2 & y_3 & \dots & y_n \\ z_1 & z_2 & z_3 & \dots & z_n \end{matrix} \tag{3.4}$$

  Vectors are normalised to unit length within the bedpostx code. For volumes in which there was no diffusion weighting, the entry should still be present, although the direction of the vector does not matter!

- bvals (with no file extension): An ASCII text file containing a list of $bvalues$ applied during each volume acquisition. The order of entries in this file must match the order of volumes in the input data and entries in the gradient directions text file. The format is:

$$b_1 \quad b_2 \quad b_3 \quad \dots \quad b_n \tag{3.5}$$

  The order of $bvals$ must match the order of data.

3.2.1.2 Outputs

bedpostx creates a new directory at the same level as the input directory called $< indir > .bedpostX$, which contains all the files you need for probabilistic tractography. Highlights are (<i> indicates the i-th fiber. It ranges from 1 to the maximum number of fibres set in the advanced options.):

- merged_th<i>samples - 4D volume - Samples from the distribution on theta.
- merged_ph<i>samples - 4D volume - Samples from the distribution on phi. theta and phi together represent the principal diffusion direction in spherical polar co-ordinates.
- merged_f<i>samples - 4D volume - Samples from the distribution on anisotropic volume fraction (see technical report).
- mean_th<i>samples - 3D Volume - Mean of distribution on theta.
- mean_ph<i>samples - 3D Volume - Mean of distribution on phi.
- mean_f<i>samples - 3D Volume - Mean of distribution on f anisotropy. Note that in each voxel, fibres are ordered according to a decreasing mean f-value.
- mean_dsamples - 3D Volume - Mean of distribution on diffusivity d.
- mean_S0samples - 3D Volume - Mean of distribution on T2w baseline signal intensity $S0$.
- dyads<i> - Mean of PDD distribution in vector form. Note that this file can be loaded into fslview for easy viewing of diffusion directions.
- nodif_brain_mask - binary mask created from nodif_brain. Copied from input directory.

3.2.2 Rumba

3.2.2.1 Inputs

The input format is similar to fomat used by bedpostx:

- data: A 4D series of data volumes in Nifti image format. This will include diffusion-weighted volumes and volume(s) with no diffusion weighting.
- nodif_brain_mask: 3D binary brain mask volume in Nifti image format.
- bvecs (with no file extension): An ASCII text file containing a list of gradient directions applied during diffusion weighted volumes. The order of entries in this file must match the order of volumes in data.

- ODFs reconstruction scheme. An ASCII text file containing a list of $Nrecons$ spatial vectors in Cartesian coordinate system. The order of entries in this file will match the order of volumes in output ODF image. $Nrecons$ is the number of pre-defined directions in the spatial reconstruction scheme. $x/y/z$ are the dimension of the original raw image volume. The format is:

$$\begin{array}{ccccc} x_1 & x_2 & x_3 & \dots & x_n \\ y_1 & y_2 & y_3 & \dots & y_n \\ z_1 & z_2 & z_3 & \dots & z_n \end{array} \tag{3.6}$$

3.2.2.2 Outputs

- odf.nii. Orientation Distribution Function (ODF) map. The file is a 4D volume, with dimension of $x * y * z * Nrecons$.
- csf.nii. Brain spinal fluid volume fraction image.
- gm.nii. Gray matter volume fraction image.

3.3 Preliminary evaluation

The performance evaluation was carried out in a computer with Ubuntu x64 version 14.04. The system's hardware consisted on a POWER8 8247-42L with 20 cores at 3.42 GHz and 64 GiB of RAM. The use case applications were compiled with -O3 and -DNDEBUG flags. Given that most of the performed computation is done with single precision floating point numbers, we have limited the SMT factor to two thread per core (40 threads in total). This is because POWER8 provides two single precision vector float point units (FPUs) per core [7]. All the evaluation results were acquired as the average of five consecutive executions.

In the following subsection, we show a preliminary evaluation performed on the CIBERSAM use cases.

3.3.1 Bedpostx and Probtrackx

A preliminary evaluation has been carried out for both Bedpostx and Probtrackx. Both applications are part of the FSL toolkit and the experiments were carried out with the original version of the source code.

On the one hand, Bedpostx was executed with an input data scenario with a file of 207 MiB, consisting on 4D data series of volumes of $148x148x60$ pixels. The purely sequential version took approximately 4 hours (4/5 minutes per slice). On the other hand, Probtrackx was executed, taken as input data the result of the execution of Bedpostx (245 Mib). In this case the overall execution time was less than 40 minutes.

Both applications are executed as a not-parallelized single process. Thus, there is a huge range for improvement.

3.3.2 Rumba

In order to maintain compatibility and soundness as much as possible, we have implemented a C++ equivalent source code using the Armadillo library [18]. Armadillo permits to represent common linear algebra operations as basic C++ operations. In this way, it is not needed to completely implement common operations like matrix multiplications. Another interesting feature of Armadillo is that it provides compatibility with existing linear algebra libraries such as OpenBlas, Intel MKL, and ATLAS. Therefore, our implementation can take advantage of multicore architectures, just deploying multiple parallel threads. The preliminary results included in this document correspond to a deployment based on Armadillo and OpenBLAS.

We have evaluated Rumba with a medium profile, consisting on 4D data series of volumes of $148x148x60$ pixels. The total execution time is 2653 seconds (44 minutes approximately). We have to highlight that the current version of OpenBLAS is not fully optimized for the POWER8 architecture. In the future we plan to evaluate Rumba using the IBM's engineering and scientific subroutine library (ESSL). This library supports BLAS subroutines, which are tuned for the POWER8 processor-based family.

3.4 Profiling

In the following subsection, we show the performance metrics, obtained from $perf$. A set of performance metrics was defined in order to characterize application behavior on the POWER8 architecture, in this case, CPU time and cache metrics are considered.

3.4.1 Bedpostx

In Listing 3.1, we show the most CPU consuming function of the xfibres source code. We identify two main time consuming functions: $compute\_signal$ and $compute\_likelihood$. These functions consume around 75% of the overall execution time.

Listing 3.1: Bedpostx performance counter. CPU metrics.

```
Samples: 740K of event 'cycles', Event count (approx.): 679516798163
+  100.00%     0.00%  xfibres  [unknown]                [k] 0000000000000000
+  100.00%     0.00%  xfibres  libc-2.19.so            [.] __libc_start_main
+  100.00%     0.00%  xfibres  libc-2.19.so            [.] 0x0000000000024d00
+  100.00%     0.00%  xfibres  xfibres                 [.] main
+   85.30%     0.01%  xfibres  xfibres                 [.] _ZN19xfibresVoxelManager7runmcmcEv
+   85.07%     0.10%  xfibres  xfibres                 [.] _ZN5FIBRE10Multifibre4jumpEb
-   50.08%     2.01%  xfibres  xfibres                 [.] _ZN5FIBRE5Fibre14compute_signalEv
   - _ZN5FIBRE5Fibre14compute_signalEv
        + 70.77% _ZN5FIBRE5Fibre14compute_signalEv
        + 29.22% _ZN5FIBRE10Multifibre4jumpEb
+   34.75%    34.19%  xfibres  libm-2.19.so           [.] cos
-   24.53%     2.32%  xfibres  xfibres                 [.] _ZN5FIBRE10Multifibre18compute_likelihoodEv
   - _ZN5FIBRE10Multifibre18compute_likelihoodEv
        + 86.07% _ZN5FIBRE10Multifibre4jumpEb
        + 13.92% _ZN5FIBRE10Multifibre18compute_likelihoodEv
```

Additionally, we observe that the function $compute\_signal$ is memory-bound given the results arise from the cache metrics (Listing 3.2).

Listing 3.2: Bedpostx performance counter. Cache metrics.

```
Available samples
182K cache-references
138K cache-misses
546K cycles
727K instructions
726K branches
107 faults
2 migrations

+   99.84%   0.00%  xfibres  [unknown]          [k] 0000000000000000
+   99.82%   0.00%  xfibres  libc-2.19.so       [.] __libc_start_main
+   99.82%   0.00%  xfibres  libc-2.19.so       [.] 0x0000000000024d00
+   99.82%   0.00%  xfibres  xfibres            [.] main
+   92.98%   0.36%  xfibres  xfibres            [.] _ZN19xfibresVoxelManager7runmcmcEv
+   91.48%   0.18%  xfibres  xfibres            [.] _ZN5FIBRE10Multifibre4jumpEb
+   46.34%   0.30%  xfibres  xfibres            [.] _ZN5FIBRE5Fibre14compute_signalEv
+   27.75%   0.05%  xfibres  libm-2.19.so       [.] exp
+   27.32%  25.84%  xfibres  libm-2.19.so       [.] __exp_finite
+   20.96%   0.40%  xfibres  xfibres            [.] _ZN5FIBRE10Multifibre18compute_likelihoodEv
+   16.47%  11.52%  xfibres  libm-2.19.so       [.] cos
+   16.20%   0.00%  xfibres  libm-2.19.so       [.] log
+   16.18%  16.04%  xfibres  libm-2.19.so       [.] __log_finite
+   12.14%   0.03%  xfibres  xfibres            [.] _ZN6NEWMAT12ColumnVectoraSERKNS_10BaseMatrixE
+   12.14%   1.08%  xfibres  xfibres            [.] _ZN6NEWMAT13GeneralMatrix2EqERKNS_10BaseMatrixENS_10MatrixTypeE
```

3.4.2 Probtrackx

In case of Probtrack, we observe that the majority of time is consumed in function $ColumnVector$. As the perf's report demonstrates, the majority of the time is consumed by this function, so our effort must be target to this region of the application.

Listing 3.3: Probtrackx performance counter. CPU metrics.

```
Samples: 5M of event 'cycles', Event count (approx.): 4706001669417
+  100.00%   0.00%  probtrackx  [unknown]          [k] 0000000000000000
+  100.00%   0.00%  probtrackx  libc-2.19.so       [.] __libc_start_main
+  100.00%   0.00%  probtrackx  libc-2.19.so       [.] 0x0000000000024d00
+  100.00%   0.00%  probtrackx  probtrackx         [.] main
+  100.00%   0.00%  probtrackx  probtrackx         [.] _Z8seedmaskv
+   99.48%   0.00%  probtrackx  probtrackx         [.] _ZN5TRACT11 Seedmanager 3runERKfS2_S2_bi
+   99.45%   0.08%  probtrackx  probtrackx         [.] _ZN5TRACT11 Seedmanager 3runERKfS2_S2 _bi RKN6NEWMAT12
    (cont.)ColumnVector E
-   94.47%   5.15%  probtrackx  probtrackx         [.] _ZN5TRACT11 Streamliner 10
    (cont.)streamlineERKfS2_S2_RKN6NEWMAT12ColumnVectorERKiS6_
   - _ZN5TRACT11Streamliner10streamlineERKfS2_S2_RKN6NEWMAT12ColumnVectorERKiS6_
      - 91.14% _ZN5TRACT11Seedmanager3runERKfS2_S2_biRKN6NEWMAT12ColumnVectorE
         + 100.00% _ZN5TRACT11Seedmanager3runERKfS2_S2_bi
      + 8.86% _ZN5TRACT11 Streamliner10streamlineERKfS2_S2_RKN6NEWMAT12ColumnVectorERKiS6_
+   26.71%   0.86%  probtrackx  probtrackx         [.] _ZN9MISCMATHS10
    (cont.)vox_to_voxERKN6NEWMAT12ColumnVectorES3_S3_RKNS0_6MatrixE
```

3.4.3 Rumba

In Listing 3.4, we show the Matlab source code of the most time consuming part of Rumba. It is important to note that this source code has been translated to an equivalent version in C++.

Listing 3.4: Initial Matlab source code of intravox_fiber_reconst_sphdeconv_rumba_sd function.

```
for i = 1:Niter
    if Dim(2) > 1
        display (['Iter->' num2str(i) 'of' num2str(Niter)]);
    end

    fODFi = fODF;
    Ratio = mBessel_ratio(n_order,Reblurred_S);
    RL_factor = KernelT*( Signal.*( Ratio ) ) ./( KernelT*(Reblurred) + eps);
    fODF = fODFi.*RL_factor;
    Reblurred = Kernel*fODF;
    Reblurred_S = ( Signal.*Reblurred) ./ sigma2;
    sigma2_i = (1/N)*sum( (Signal.^2 + Reblurred.^2)/2 - (sigma2.*Reblurred_S).*Ratio, 1) ./ n_order;
    sigma2_i = min((1/10)^2, max(sigma2_i,(1/50)^2));
    sigma2 = repmat(sigma2_i,[N, 1]);
end
```

We have identified the most time consuming region of the source code. As we show in Listing 3.5,

In most of the cases, we set up the number of performed iterations to 300 (this number determines the final quality of the simulation). We estimated that this region needs 900 matrix multiplications and 900 product-wise multiplications. Additionally, we identify a possible reduction loop inside the iteration loop. In order to accelerate this region, we need to parallelize all the linear algebra operations.

The code shown in Listing 3.5 is executed for each of the input volume's slices, thus the approximate number of matrix multiplications could be larger than 9,000.

Listing 3.5: C++ version translated from the code presented in Listing 3.4. Operator * represents matrix multiplication and operation % represents product-wise multiplications (element by element).

```
    for (auto i = 0; i < Niter; ++i) {
        T sigma2_i;
        fODFi = fODF;
        Ratio = mBessel_ratio<T>(n_order,Reblurred_S);

        RL_factor = KernelT * (Signal % Ratio) / ((KernelT * Reblurred) + std :: numeric_limits<double>::epsilon());

        fODF = fODFi % RL_factor;
        Reblurred = Kernel * fODF;
        Reblurred_S = (Signal % Reblurred) / sigma2;

        T sum = 0.0;
        for (size_t i = 0; i < Signal.n_rows; ++i) {
            for (size_t j = 0; j < Signal.n_cols; ++j) {
                sum += (std :: pow<T>(Signal(i,j),2) + std :: pow<T>(Reblurred(i,j),2)/2 - (sigma2*Reblurred_S(i,j)) * Ratio(i,j));
            }
        }
        sigma2_i =(1.0/N) * sum / n_order;
        sigma2 = std :: min<T>(std::pow<T>(1.0/10.0,2), std :: max<T>(sigma2_i, std::pow<T>(1.0/50.0,2)));
    }
```

3.5 Requirements

The following requirements have been identified for all the CIBERSAM's use cases:

- General. The implemented source code has to be implemented in C++. The FSL toolkit is already implemented in C++, so we need to work in the same

programming language. In case of the Rumba use case, we part from existing source code in Matlab. This source code will be used as a requirement specification.

- Hardware platform. The implemented software must be executed on both x86 and POWER (OpenPower) platforms. Nowadays, FSL and Rumba prototypes work on both x86 and POWER platforms.
- Processing Units. The source code is highly dependent of linear algebra. Thus, accelerating matrix multiplications is a key point of both use cases. As establish in the RePhrase project, the code must be accelerated by using multicore architectures and GPUs.
- Operating system. Given the customer range, the solution have to support both Windows and Linux operating systems.
- Programming Technology. Use cases have as requirement CUDA, C++11/14/17 features, OpenMP, and Intel TBB.

4 Stochastic Local Search

4.1 Introduction

In order to increase competitiveness of any industry, effective utilization of all kind of resources, from machines and raw material to energy and human resources, and highly optimized process cycles are crucial. Examples are minimization of waste in roll cutting in clothing [11] or paper industry [3], optimization of control loops in chemical plants or oil refineries or optimization in supply chain management or logistics, just to name a few. This use case deals with an optimization problem encountered in the slitting of metal sheets used in the production of electrical transformers. The problem is a generalized version of the so-called 1/V/V/R cutting-stock problem, since the objective corresponds to appropriately placing a set of metal stripes (bands) into a set of available metal coils, so that the overall metal waste is minimized.

Several methodologies have been developed to address cutting-stock problems or bin-packing problems, including linear-programming (LP) based approaches [5, 14] and heuristic-based approaches based on dynamic programming [10]. However, the complexity of the optimization problem as well as the large number of (potentially nonlinear) constraints cannot effectively be addressed through LP-based approximations. At the same time, experts knowledge is required for effectively implementing heuristic-based approaches as in [10]. To this end, we focus on *stochastic-local-search algorithms* [15] for addressing a generalized class of such cutting-stock problems, since a) no explicit assumptions are imposed regarding the form of the constraints, and b) the design of such algorithms does not necessarily require experts knowledge.

The performance of stochastic-local-search algorithms may not necessarily be robust to the specifics of the optimization problem, while convergence to local minima cannot easily be excluded. To this end, it is usually required that several forms of *diversification strategies* (cf., [15]) need to be implemented, including a) *reprocessing candidate solutions* from earlier optimization stages, and b) *experimenting alternative processing paths.* But since introducing such strategies increases the runtime of the optimization process a lot. Parallelization of this algorithm is essential to compensate the additional runtime and to increase the overall performance

of the result.

4.2 Use Case Description

In this use case, we are particularly concerned with a generalized version of the so-called 1/V/V/R cutting-stock problem [13]. Cutting-stock problems are encountered often in industrial environments and the ability to address them efficiently usually results in large economic benefits.

Such cutting-stock optimization is encountered in the electrical transformers' industry as described in detail in [10]. In particular, in the production of the core of an electrical transformer, great quantities of silicon-steel sheets are required which could reach in weight up to 300 tones for large transformers. The silicon-steel sheets required vary in width and need to be slit from available rolls of material. The problem can be translated in a straightforward manner into a classical cutting stock problem [10].

4.2.1 Cutting-stock problem (background)

This section briefly describes the one-dimensional cutting-stock optimization problem. There exists a set of available *objects* (or *rolls*) of material, denoted by $\mathcal{I} \doteq \{1, 2, ..., m\}$. Each of these objects $i \in \mathcal{I}$ is characterized by its width b_i, its length ℓ_i and its density d_i, $i \in \mathcal{I}$. Let also w_i denote the overall weight of roll i. In Figure 4.1 one such roll is depicted.

We are also provided with a set of *items* (or *bands*) of certain types $\mathcal{J} \doteq \{1, 2, ..., n\}$, each of which is characterized by its width b_j and its desired weight w_j. The purpose of the optimization is to compute an assignment of the desired bands into a set of available rolls so that the total weight of the rolls used is minimized.

Formally, let us denote $x_{ij} \in \mathbb{Z}_+$ as the number of bands of type j assigned to item i. The objective is to find an assignment $X \doteq \{x_{ij}\}_{i,j}$ of the desired bands into the set of available rolls, so that:

1. the overall weight of each band type j exceeds its desired weight w_j, i.e.,

$$b_j \sum_{i=1}^{m} x_{ij} \ell_i d_i \geq w_j, \tag{4.1}$$

2. the sum of the bands assigned to each roll i does not exceed the width of the roll, b_i, while at the same time the residual band, denoted

$$r_i(X) \doteq b_i - \sum_{j=1}^{n} x_{ij} b_j, \tag{4.2}$$

should always be within a finite set $\mathcal{R}$ of allowable residual widths.

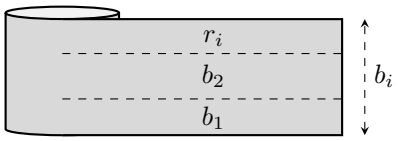

Figure 4.1: Roll slitting.

Both of the above constraints are hard constraints and need to be satisfied for any assignment. Unfortunately, in most but trivial cases, there might be a multiplicity of admissible assignments, each of which might be utilizing a different subset of rolls $\mathcal{I}$. We wish to minimize the overall weight of the rolls utilized by an assignment, i.e., we wish to address the following optimization problem:

$$\min_{X \in \mathbb{Z}_+^{m \times n}} \sum_{i \in \mathcal{I}} w_i \mathbb{I}_{\{\exists j \in \mathcal{J} : x_{ij} > 0\}}, \tag{4.3}$$

where

$$\mathbb{I}_A \doteq \begin{cases} 1 & \text{if } A = \texttt{true}, \\ 0 & \text{else.} \end{cases} \tag{4.4}$$

In other words, we would like to penalize the weight of the rolls which are slit.

The objective function (4.3) subject to the job-admissibility constraint (4.1) and the rest-width-admissibility constraint (4.2) formulate the so-called *(one- dimensional) cutting-stock problem*.

Alternative objective criteria may be considered (e.g., minimization of trim loss, as considered in [13], or minimization of rolls needed by the assignment when the objects are identical, as considered by [5]).

4.2.2 Objective

In practical scenarios, as in the case of electrical transformers industry [10], additional constraints may also be imposed (due, e.g., to cutters specifications, transformer final specifications, etc.). Given that most of such constraints may be nonlinear in nature, traditional methods based on linear programming (as described in [5]) are not appropriate for such problems. Furthermore, a large number of constraints may reduce dramatically the set of feasible solutions, making the search over optimal solutions even harder.

To this end, stochastic-based approaches have been considered to address such complex optimization problems (e.g., the stochastic-local-search algorithms discussed in [15]). Such approaches consist of a sequence of (local) modification steps onto the current candidate solution so that the overall objective function is reduced. The advantage of such methods is the ability to provide suboptimal solutions within reasonable execution times. On the other hand, due to their stochastic nature, the resulting performance may vary, depending on the details of the optimization problem, the running time and the details of the stochastic search method.

Due to the performance variability, it is often required that we provide a set of *diversification strategies* that decrease the probability of converging to a local optimum, or, equivalently, increase the probability of converging to the global optimum. One example of such diversification strategies includes the ability to periodically *reprocess* candidate solutions starting from earlier stages of the optimization. Another diversification strategy may also include the ability to *experiment* alternative processing paths starting from the same candidate solution.

The objective of this paper is to explore the utility of *parallelization* for improving the performance of stochastic-local-search algorithms in the context of hard-combinatorial problems as the cutting-stock problem presented above. In particular, we wish to explore the utility of parallelization in improving the effect of the diversification strategies a) *reprocessing candidate solutions*, and b) *experimenting alternative processing paths* onto the optimization performance.

4.3 Algorithm Description

In this section, we propose a framework for implementing a stochastic local-search approach for addressing such cutting-stock problems.

More specifically, the proposed framework consists of the following building blocks:

- $\mathfrak{P}$=`init(`π`);`
- X=`optimize(`$\mathfrak{P}$`);`
- $\mathfrak{P}$=`filter(`X`,`$\mathfrak{P}$`);`
- `terminate(`$\mathfrak{P}$`,timer()).`

The role of the `init(`π`)` function is the establishment of candidate solutions satisfying (at least) constraint (4.1), but not necessarily all constraints. The large number of constraints in a cutting-stock problem imposes difficulties in finding even feasible solutions. In such cases, such initialization phase is rather important in a) computing feasible solutions, b) reducing the overall optimization time, or equivalently c) improving the overall performance. This initialization phase may correspond to standard First-Fit-Decreasing (cf. [21, Section 3.3]) type of algorithms, whose goal is to simply allocate the required items/bands onto the available objects so that the job-admissibility constraint (4.1) is satisfied. However, in most cases, satisfying even constraint (4.1) may be rather challenging, thus more sophisticated initialization algorithms may be required. An initial set $\mathfrak{P}$ of candidate solutions is defined as the output of this initialization phase.

The `optimize(`$\mathfrak{P}$`)` function constitutes the core of the overall optimization framework. Its role is the execution of appropriate (local) modification steps (called *operations*) onto the candidate solutions $X \in \mathfrak{P}$, accompanied with appropriate random perturbations. The role of these operations is to search for a) candidate

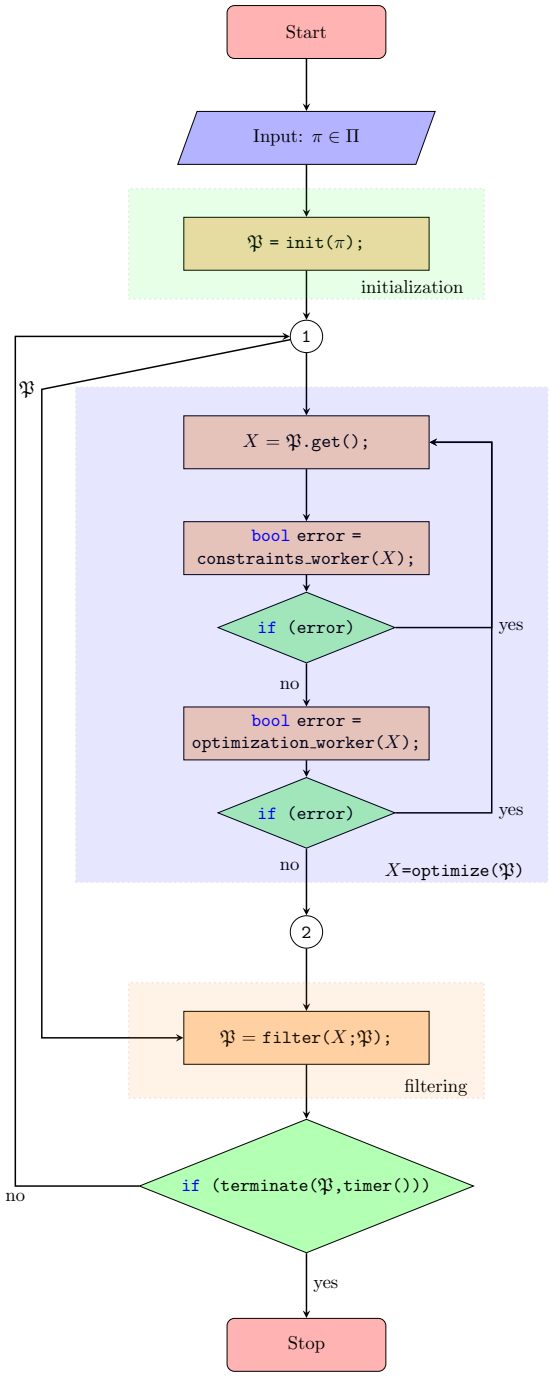

Figure 4.2: Architecture of optimization algorithm.

solutions which satisfy all imposed constraints, and b) candidate solutions which improve the cost of (4.3). Responsible for the execution of these improvement

steps and/or perturbations of the existing candidate solutions are the working units, briefly called `workers`. Each worker serves a distinctive role (e.g., creating admissible solutions with respect to a constraint or reducing the cost function or locally perturbing the solution). Designing the operation of such workers requires a careful design of stochastic-local search strategies.

One of the main drawbacks of stochastic-local search algorithms is the fact that convergence to local minima may occur with high probability (see, e.g., [15, Section 5.2]). This is particularly evident as the number of constraints increases requiring for additional treatment. To this end, a careful design of so-called *diversification strategies* (cf., [15, Section 5.2]) is required. An example of such diversification strategies includes the ability to reprocess candidate solutions from earlier processing stages, which will increase the probability of escaping from local optima. Such diversification strategies can be designed within the `filter(`X`,`$\mathfrak{P}$`)`. Its primarily goal is to determine the population of the pool $\mathfrak{P}$ which will feed back to the `optimize()` function. Its impact on the overall behavior of the optimization algorithm is equally important to the role of the `optimize()` operations.

4.4 Parallelization Possibilities

The basis for the parallelization is an already working sequential implementation of the whole optimization program including the algorithm described above. The core construct of the optimization part as shown in Figure 4.2 is the main loop that iterates over the candidate solutions in a pool. In every iteration one candidate solution is picked from the pool of solutions and first passed to the *optimize* function, where the demanded constraints are applied and local optimization steps are executed, and afterwards to the *filter* function, where the on the one hand the current candidate solution is inspected and the whole optimization process is observed and decided whether to continue or terminate.

Investigation of the static code structure and the runtime behavior identified following parallelization possibilities:

- Main loop: The main loop is the part of the code where the vast bulk of execution time is spent. Furthermore it turned out that this part also has the greatest potential for parallelization since the optimize function can be applied on all candidate solutions simultaneously as the solutions are independent from each other and do not share any state that could lead to race conditions. An accordingly modified figure of the architecture is depicted in Figure 4.3

- Worker: Within the optimize function the *contraints_worker* and *optimization_worker* bear further potential for parallelization. But the operations that

are applied to the slitting plan iteratively in a loop are not completely independent since they work on partly shared data structures.

- Filter: The filter method also provides parallelization potential. But similar to the afore mentioned workers the loop iterations within the filter are not completely independent and have to share common data structures.

The parallelization is subject to following general constraints:

- Since the code has already been very complex, additional complexity introduced by parallelization should be as less a possible.
- Further development and maintenance will be done by non parallelization-experts, which is why code changes should be easy to understand and restricted to a small code area.
- Nevertheless, the parallelization model has to provide flexibility; though the algorithm is iterative, the number of iterations is not known at the beginning. Termination of the optimization process is based on different conditions that might be fulfilled anytime during the the process.

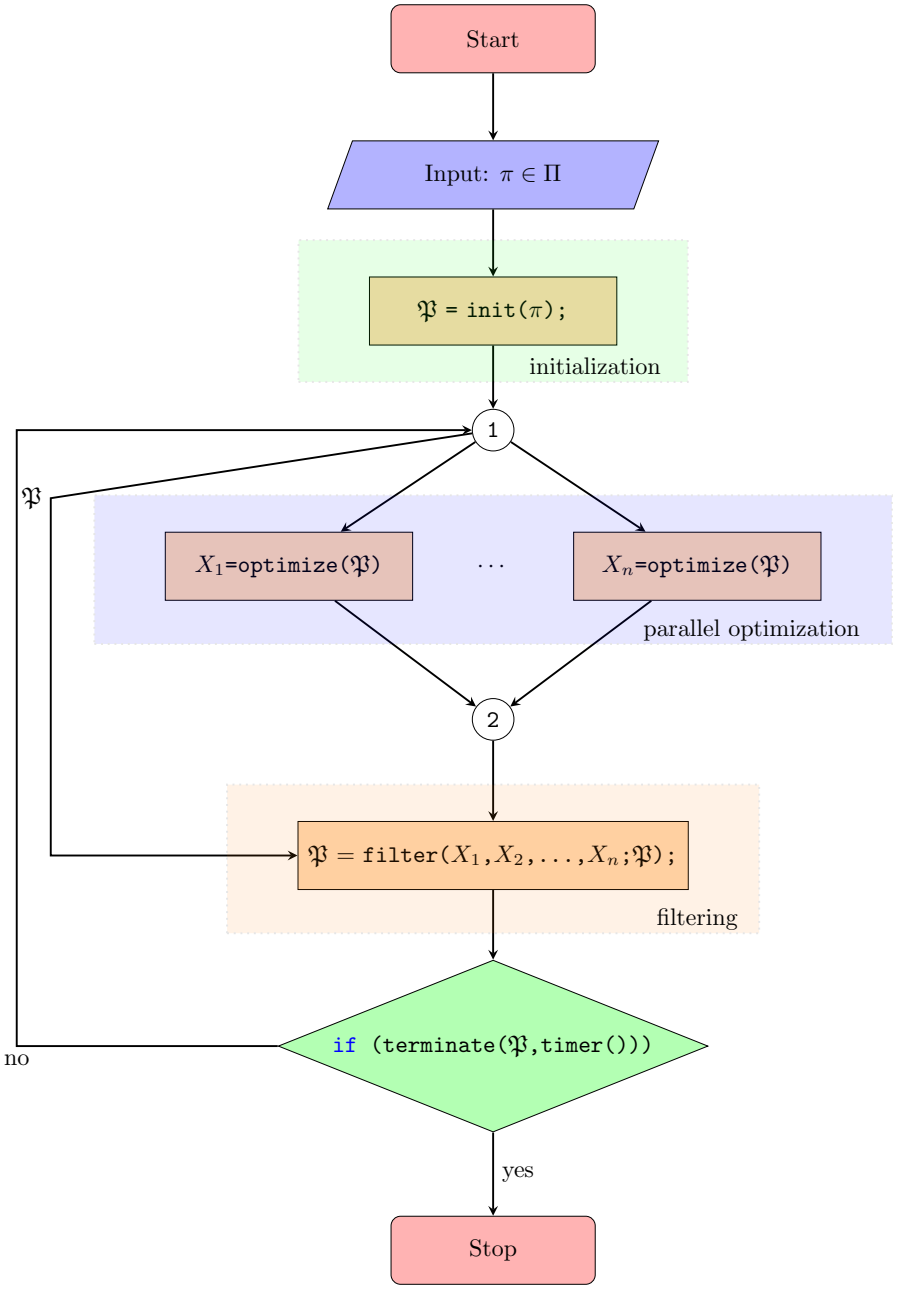

Figure 4.3: Architecture of parallelized optimization algorithm.

5 Weather Forecast

5.1 Use Case Description

5.1.1 Motivation

Many systems exist that are heavily influenced by environmental parameters like temperature, precipitation or solar radiation. One common example is the road network. Snowfall or freeze pose safety risks that have to be compensated by road maintenance companies. But to be able to plan their measures than just react on changing conditions, very precise weather forecasts are essential. Similar is true for renewable energy sources. Planing energy efficiency of wind farms and solar plants depends on accurate local prediction of wind and global radiation.

To achieve planing reliability, forecasts need to be of high quality in prediction of certain environmental variables and of high *spatial* and *temporal* resolution. Typical weather models as used by e.g. the German Weather Service have a very rough spatial resolution. They do not take local topographic characteristics into account as can be found in the foothills of the Alps or in the Northern Alps. But these specifics have a great impact on local weather conditions. To optimize location-dependent weather forecasts these characteristics have to be included into the prediction models.

5.1.2 Automatic Local-dependent Weather Forecast

As stated above global weather forecasting models lack of spatial and temporal resolution. Figure 5.1 gives an overview of the spatiotemporal resolution of different weather models [6]. It shows the classification of forecasting models based on spatial resolution of input data and temporal resolution of output or foreseen data. Such a resolution is too rough to accurately predict environmental variables for a specific location since local influences (e.g. topographical characteristics) can lead to deviations between global predictions and real weather conditions in small areas. To get a better resolution for local-dependent forecasts new local weather models based on global models have to be developed.

A common approach is to oppose historical global predictions of environmental variables to real measurements of local weather stations over a long time period [19]. On this comparison as a basis statistical and learning methods (data

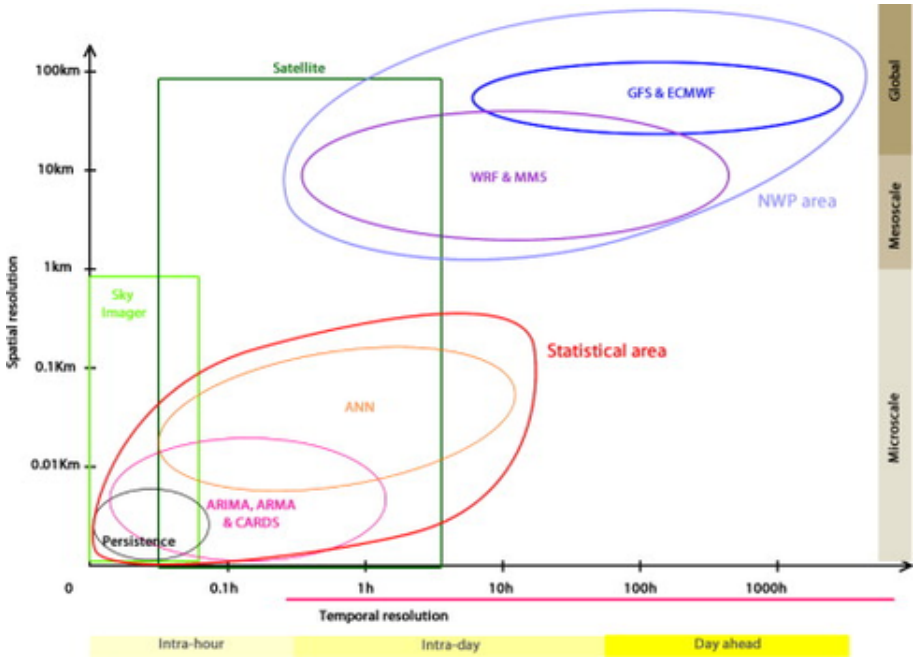

Figure 5.1: Classification of model based on spatial and temporal resolution.

mining, machine learning) are applied to minimize deviation of global predictions and measurements [9,20]. Primary models $M_{o,f}$ have to be found that at time t for a location o and a prediction period f are able to predict a certain meteorological variable $P(t+f)$ for time $t+f$ based on available data $G(t)$ at time t from a global weather model:

$$P(t+f) = M_{o,f}\left(G(t)\right) \tag{5.1}$$

Different meteorological variables show different statistical characteristics. E.g. temperature has a continuous development, spatial as well as temporal. Close locations tend to have similar temperatures and temperature also doesn't change abruptly. Precipitation in contrast tends to change fast and close locations can have very different precipitation rates. Not every forecast model is equally suitable for every variable and therefor it is very important to select the most appropriate model for each one.

5.2 Current System: BlueForecast

In a long-term research cooperation between *Bluesky*[1] and *Software Competence Center Hagenberg* the weather forecast system named *BlueForecast* has been developed. The system uses global weather data mainly from the *Global Forecast*

[1] www.blueskywetter.at

System (GFS), the German Weather Service (DWD) and measurement data from local weather stations. To build the forecast models the system utilizes a pool of different methods:

- Feature selection: selection of relevant parameters
- Model generation: ridge regression, decision-/ regression trees, fuzzy-logic based methods, neuronal networks, kernel methods
- Model selection: cross validation, Leave-One-Out

The used methods are very powerful and able to achieve a good prediction accuracy. But to reach the aimed high quality expertise in meteorology is essential and needs to be integrated into the process of model generation.

- Feature selection/generation: Experts help selecting which global environmental variables and combinations of them should be used within the models.
- Model evaluation: Experts examine the structural level of the models and check for implausibilities. The results are integrated into the process of model generation.
- Model initialization: E.g. based on meteorological rules are formulated as fuzzy rule set and optimized on the data basis; definition of similarity metrics (kernel) based on expert knowledge to be integrated in to learning methods like support vector machines.

5.2.1 Global Weather Model

The mainly used global weather model is the *Global Forecast System (GFS)*, a numerical weather prediction model produced by the National Centers for Environmental Prediction (NCEP). It is a coupled model of four separate models (an atmosphere model, an ocean model, a land/soil model, and a sea ice model). Dozens of atmospheric and land-soil variables are available for the whole globe. The base horizontal resolution is 13 kilometers between the grid points for predictions up to 10 days and 27 kilometers for predictions from 10 to 16 days. The model is run four times a day and can be downloaded through the NOAA National Operational Model Archive and Distribution System (NOMADS). GFS data is not copyrighted and is available for free in the public domain under provisions of U.S. law. Because of this, the model serves as the basis for the forecasts of numerous private, commercial and foreign weather companies.

Figure 5.2 shows an example of prediction map of the GFS, in this case the predicted 6-hour averaged precipitation rate.

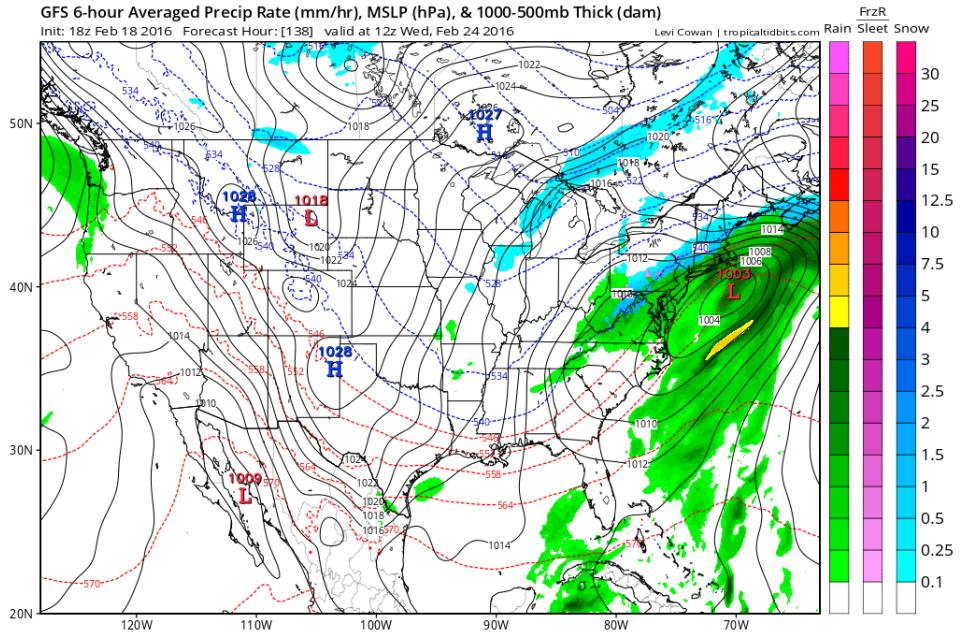

Figure 5.2: GFS 6-hour averaged precipitation rate (mm/hr).

5.2.2 Architecture

Figure 5.3 shows the coarse architecture of the BlueForecast system. The core of the system is a web-server application consisting of following components:

- Data Provider: Reads incoming data from GFS, DWD (German Weather Service) and the local measurements into internal data structures
- Conversion Logic: Gets weather forecast data and measurements from the Data Provider and performs different preprocessing steps for further usage in model generation. All data is stored in a database instance.
- mlf + Mathematica: The mlf (Machine Learning Framework for Mathematica [17]) provides a pool of statistical and machine learning methods used in the model generation process.
- Model Generator: Learns all the models for the different configured location, time ranges and meteorological variables.
- Model Application: Runs the learned models and creates the forecasts that are stored in a database.
- Forecast Provider: The forecast provider is a service that waits for request for weather forecasts.

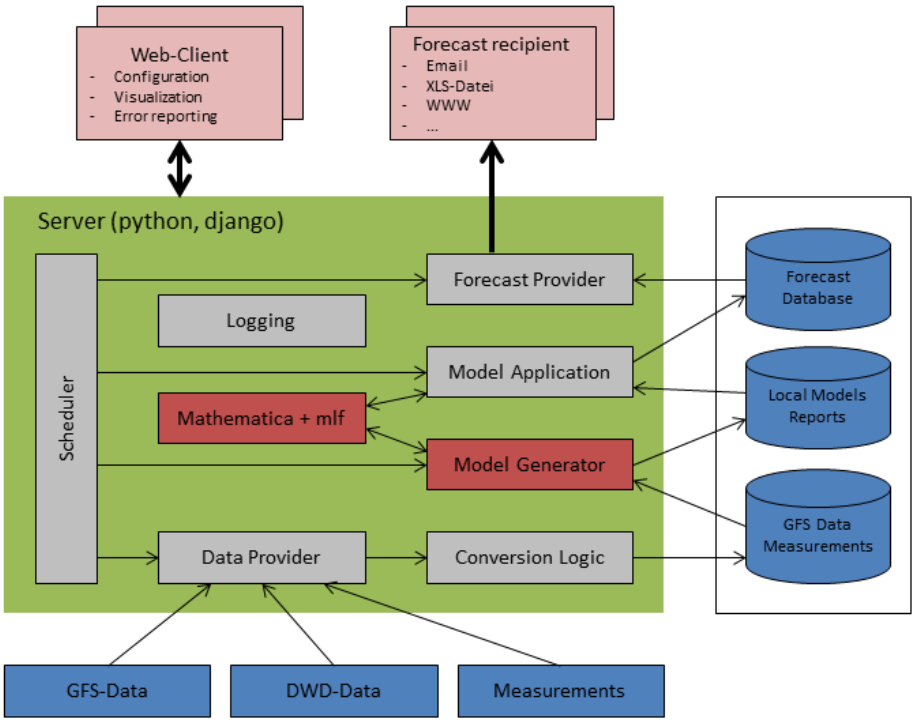

Figure 5.3: Blue Forecast system architecture.

- Scheduler: The scheduler is responsible for scheduling all server tasks (data conversion, model generation, model application, forecast requests).

- Logging: Logs all events that occur on the server (e.g. data income, model generation, ...)

5.2.3 Accuracy

The current system already allows accurate forecasts of temperature and precipitation for selected locations. It has been evaluated against an established reference model currently in use at BlueSky over a period of four month.

5.2.3.1 Precipitation Prediction

Forecast if precipitation on the next day (12:00 - 18:00, 36-hour forecast) will be more or less than a certain threshold (mm). Table 5.1 and Table 5.2 show the evaluation results for Aigen im Ennstal and Feuerkogel for different thresholds. In 9 out of 10 cases BlueForecast outperforms the reference model.

Threshold	Model	Accuracy
0	BlueForecast	83.9
0	Reference model	81.4
1	BlueForecast	80.7
1	Reference model	70.2
2	BlueForecast	85.9
2	Reference model	78.0
5	BlueForecast	78.8
5	Reference model	62.9
10	BlueForecast	45.2
10	Reference model	48.7

Table 5.1: Accuracy of Precipitation Forecast (Aigen im Ennstal)

Threshold	Model	Accuracy
0	BlueForecast	86.2
0	Reference model	80.6
1	BlueForecast	83.2
1	Reference model	81.7
2	BlueForecast	84.3
2	Reference model	82.2
5	BlueForecast	79.5
5	Reference model	73.1
10	BlueForecast	73.1
10	Reference model	68.9

Table 5.2: Accuracy of Precipitation Forecast (Feuerkogel)

5.2.3.2 Temperature Prediction

Forecast of temperature at noon on the next day in Linz and Aigen im Ennstal (36-hour forecast). Table 5.3 shows the average deviation between prediction and measured values. Again BlueForecast outperforms the reference model.

	BlueForecast	Reference model
Linz	1.1°C	1.4°C
Aigen im Ennstal	1.8°C	2.2°C

Table 5.3: Accuracy of Temperature Forecast

5.3 System Extension

The current system provides forecast models for a few selected locations and environmental variables (temperature and precipitation) and is able to make predictions for a forecast period of 36 hours. Since the results for these models are very promising the system is going to be extended.

- The system will be enlarged to support predictions of ten different environmental variables for 1000 locations. Furthermore the forecast period range will be extended. Forecasts will be provided for up to 180 hours (280 hours for selected variables) split into time ranges of 3 hours. The first model evaluation showed that the best results can be achieved if an individual model is trained for every location, every variable and every forecast period. For the planned extension (1000 locations, 10 variables, 180 hours) this will lead to 600.000 different models.
- The current method pool will be extended with new methods. Especially deep learning models have been successfully applied in weather forecast models and are going to be added to the method pool.
- Furthermore the used tool chain is going to be cleared. Currently the main parts are written in python and C++ and the machine learning methods are based on Mathematica. This part is going to be replaced with python and C++ implementations.

5.3.1 Deep Learning Model

Deep learning has already been a topic of research for many years. It has been inspired by the way the human brain works. E.g. the well-studied visual cortex is known to be built up of different areas of the brain that are connected hierarchically, where each level of this feature hierarchy represents a different level of abstraction of the input signal. The higher up in the hierarchy the more abstract the features are,

defined in terms of the lower-level ones. Deep architectures are composed of levels of non-linear functions allowing them to compactly represent highly non-linear and highly-varying functions. Learning such representations allows a system to learn complex functions mapping the input to the output directly from data without depending completely on human-crafted features which are often impossible to created due to the high complexity of the input data.

But training such deep architectures has not been very successful at the beginning. First positive results have been achieved with rather shallow architectures consisting of one or two hidden layers, but results got poorer as depth increases. Real breakthroughs have been achieved with the introduction of Deep Belief Networks and auto-encoder based architectures. Since that time many improvements in training methods have been made that lead to much better performance of deep architectures.

Another problem of training deep models is the computational intensity that makes training of models with many broad layers and big amount of input data very time consuming. Use of parallel hardware, especially GPUs, made the training much faster and more effective so that machine learning algorithms based on deep architectures now outperform traditional approaches in many areas (e.g. natural language processing or computer vision). Recently deep learning has also been applied in weather forecasting with very promising results [4, 12, 16].

Based on the emerging success of deep learning in many different areas and recently also in weather predictions the current pool of machine learning algorithms used in BlueForecast will be extended with these models. It is planned to develop a framework that allows to experiment with different kinds of deep models, different architectures and ensembles of machine learning algorithms and to integrate it into the current system.

5.3.1.1 Caffe Framework

In the last years many different frameworks for deep learning have been developed. Each with its own strengths and weaknesses and specializations for particular kinds of deep models (e.g Torch, DeepLearning4j, cuDNN, Keras or Caffe). The selected framework has to fulfill following requirements: common used programming language (C++), modularity, support of different kinds of deep learning models and extensibility for new models. The framework of our choice is Caffe [2]. It has been developed by the Berkeley Vision and Learning Center (BVLC) and by community contributors. Caffe has been primarily developed for computer vision problems and therefor provides already support for GPU acceleration. Parallelization on multi-/many core systems is only available if the used BLAS (e.g. OpenBlas) supports this.

[2] caffe.berkeleyvision.org

5.4 Parallelization Possibilities

To provide the full functionality of the extended system as described above, about 600.000 models have to be trained and every day when new GFS data (about 2-3 Gb/day) is available these models have to be applied to calculate the new forecasts. It is obvious that this is computationally very intensive and not possible without parallelization.

The possibilities for parallelization are manifold, reaching from a very coarse level (e.g. train models in parallel) to a very fine grained (e.g. BLAS operations). In this use cases the parallelization will be constrained to the new developed deep learning framework. Currently following parallelization possibilities have been identified:

- Caffe framework: At its current state the Caffe framework only GPU acceleration and parallelization using multithreaded BLAS implementations are available. But there are several possibilities in the framework where CPU parallelization is possible. In a first examination several for-loops in the Solver class have been identified. RePhrase technologies especially refactoring and pattern discovery should help to find more possibilities. Verification and testing tools should assure that no errors will be introduced into the framework.

- Data preprocessing: Before the data from the global forecast model and the data from the measurement stations can be used, some preprocessing steps have to be performed. Some of these steps might be independent from each other and can be done in parallel or the same step can be performed on different portions of data.

- Model training: Since all models are almost independent of each other it is possible to train more than one in parallel. In the current implementation the potential for parallelization is limited since models for geographical close locations share some temporal data structures which causes race conditions. Therefor parallelization is only possible for distant locations.

- Model/feature selection: Evaluation of different models and features can also be done in parallel.

Due to limited resources efficient scheduling of all these parallel tasks is very important. Different tasks have different priorities (model training vs. model application) and the user is always able to interfere and define new tasks (e.g. schedule new models for training). This makes the system highly dynamic and the number threads and their priority might always change. The dynamic scheduling and dynamic compilation possibilities that are going to be developed in RePhrase should help for optimal utilization of the provided hardware resources.

6 Conclusion

In this deliverable, we have described the industrial use cases that will be used in the **RePhrase** project to evaluate tools, technologies and methodology developed in other technical workpackages (WP2-WP5). The selected use cases are from distinct domains and have different performance goals – retrieving the same results in shorter time, obtaining better results in the same time or achieve real-time response time. This ensures that the **RePhrase** methodology and tools will be broadly applicable.

Source code for most of the use cases is already available, enabling us to test refactoring, testing, verification and dynamic adaptivity tools on them in near future. Other use cases, e.g. weather forecast, are not yet fully implemented, which gives us the opportunity to evaluate the design parts of the **RePhrase** methodology and to see how successfuly it deals with the new parallel applications. The common requirement for all applications is that the results they produce have to be correct, which is ensured by the test and verification parts of the **RePhrase** toolchain.

The selected uses cases have been taken from the ongoing projects at the WP6 partners. In the course of the project, requirements and specifications of the use cases might change, which will be reflected upon in other technical work packages and will drive further development of the tools and methodology.

Bibliography

[1] TEJ Behrens, H Johansen Berg, Saad Jbabdi, MFS Rushworth, and MW Woolrich. Probabilistic diffusion tractography with multiple fibre orientations: What can we gain? *Neuroimage*, 34(1):144–155, 2007.

[2] Erick J Canales-Rodríguez, Alessandro Daducci, Stamatios N Sotiropoulos, Emmanuel Caruyer, Santiago Aja-Fernández, Joaquim Radua, Jesús M Yurramendi Mendizabal, Yasser Iturria-Medina, Lester Melie-García, Yasser Alemán-Gómez, et al. Spherical deconvolution of multichannel diffusion MRI data with non-Gaussian noise models and spatial regularization. *PloS one*, 10(10):e0138910, 2015.

[3] M Helena Correia, José F Oliveira, and J Soeiro Ferreira. Reel and sheet cutting at a paper mill. *Computers & Operations Research*, 31(8):1223–1243, 2004.

[4] Mladen Dalto, Jadranko Matusko, and Mario Vasak. Deep neural networks for ultra-short-term wind forecasting. In *Industrial Technology (ICIT), 2015 IEEE International Conference on*, pages 1657–1663. IEEE, 2015.

[5] JM Valério De Carvalho. Lp models for bin packing and cutting stock problems. *European Journal of Operational Research*, 141(2):253–273, 2002.

[6] Maimouna Diagne, Mathieu David, Philippe Lauret, John Boland, and Nicolas Schmutz. Review of solar irradiance forecasting methods and a proposition for small-scale insular grids. *Renewable and Sustainable Energy Reviews*, 27:65–76, 2013.

[7] Dino Quintero, Dino and Li, Wei and dos Santos Moschetta, Wainer and Faria de Oliveira, Mauricio and Pozdneed, Alexander. *Implementing an IBM High-Performance Computing Solution on IBM POWER8*. IBM Redbooks, 9 2015.

[8] D.C. Van Essen, K. Ugurbil, E. Auerbach, D. Barch, T.E.J. Behrens, R. Bucholz, A. Chang, L. Chen, M. Corbetta, S.W. Curtiss, S. Della Penna, D. Feinberg, M.F. Glasser, N. Harel, A.C. Heath, L. Larson-Prior, D. Marcus, G. Michalareas, S. Moeller, R. Oostenveld, S.E. Petersen, F. Prior, B.L.

Schlaggar, S.M. Smith, A.Z. Snyder, J. Xu, and E. Yacoub. The Human Connectome Project: A data acquisition perspective. *NeuroImage*, 62(4):2222 – 2231, 2012.

[9] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics Springer, Berlin, 2001.

[10] Alois Gerstl and Stefan E Karisch. Cost optimization for the slitting of core laminations for power transformers. *Annals of Operations Research*, 69:157–169, 1997.

[11] Miro Gradišar, Jože Jesenko, and Gortan Resinovič. Optimization of roll cutting in clothing industry. *Computers & Operations Research*, 24(10):945–953, 1997.

[12] Aditya Grover, Ashish Kapoor, and Eric Horvitz. A deep hybrid model for weather forecasting. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 379–386. ACM, 2015.

[13] RW HAESSLER and PE SWEENEY. Cutting stock problems and solution procedures. *European journal of operational research*, 54(2):141–150, 1991.

[14] Christoph Helmberg. Cutting aluminium coils with high length variabilities. *Annals of Operations Research*, 57(1):175–189, 1995.

[15] Holger H Hoos and Thomas Stützle. *Stochastic local search: Foundations & applications*. Elsevier, 2004.

[16] Moinul Hossain, Banafsheh Rekabdar, Sushil J Louis, and Sergiu Dascalu. Forecasting the weather of nevada: A deep learning approach. In *Neural Networks (IJCNN), 2015 International Joint Conference on*, pages 1–6. IEEE, 2015.

[17] Thomas Natschläger, Felix Kossak, and Mario Drobics. Extracting knowledge and computable models from data-needs, expectations, and experience. In *Fuzzy Systems, 2004. Proceedings. 2004 IEEE International Conference on*, volume 1, pages 493–498. IEEE, 2004.

[18] Conrad Sanderson. Armadillo: An open source C++ linear algebra library for fast prototyping and computationally intensive experiments. 2010.

[19] Z Sokol. Mos based forecast of 6-hourly area precipitation. *Studia Geophysica et Geodaetica*, 50(1):105–120, 2006.

[20] Daniel S Wilks. *Statistical methods in the atmospheric sciences*, volume 100. Academic press, 2011.

[21] David P Williamson and David B Shmoys. *The design of approximation algorithms*. Cambridge university press, 2011.