# Patterns in Machine Learning: A new parallelization workflow for Machine Learning methods

Holger Schöner and Michael Roßbory

Software Competence Center Hagenberg (SCCH) GmbH, Hagenberg, Austria.

**Email:** {holger.schoener,michael.rossbory}@scch.at

**Abstract.** Parallelization of Machine Learning methods is an active research area, fuelled by the need for acceleration of complex computations, and the constant growth of numbers of samples and features in available data sets. Because several Machine Learning methods are general in the sense that they can be reused again and again for new learning tasks, it is common to collect these methods in libraries, e.g. the library *mlpp* at SCCH. Such libraries are intended to be used by several users on different hardware platforms. As a result, it is important that their parallelization does not introduce dependence on a restricted set of deployment environments. The ParaPhrase approach, besides having advantages in the modelling of parallelism and the parallelization process, promises to provide the needed flexibility with respect to supported hardware, by targeting multicore machines, distributed clusters, and hardware accelerators like GPGPUs. The process of parallelizing one method in *mlpp*, Coordinate Descent, is illustrated in the following.

## 1 Introduction

Automation of manufacturing processes is continually growing. With it, the potential for collection and usage of data for optimization of these processes also increases. Machine Learning is an important technology for performing such analysis and optimization. With growing opportunities and sizes of data sets the demands on analysis systems rise as well, and parallelization targeting computation clusters or GPUs is an inviting direction for fulfilling these demands. Parallelization already is an active research area in Machine Learning [2,5,7], but the main systems/libraries so far have no or limited capabilities in this respect.

To meet the need for better performance of Machine Learning (ML) methods, SCCH in the context of the ParaPhrase [6] project parallelizes existing ML methods used in projects with partners, and extends the collection of available methods by new and parallel approaches. The resulting sequential and parallelized versions of learning algorithms are collected in the library “mlpp” (Machine Learning using Parallel Patterns). The parallelization is done using the pattern based approach pursued by the ParaPhrase project, but the library will also contain versions of algorithms parallelized using other parallelization frameworks, for comparison of performance and implementation complexity.

In this contribution, we present a typical data analysis problem, and describe parallelization, advantages, and issues of the ParaPhrase methodology.

## 2 Machine Learning use case

In the use case, we implement a causality detection framework for analyzing waste water processing data, provided by a partner company. The data is collected in a plant processing water from industrial as well as commercial and residential neighborhoods. It contains time series data for 6000 features, collected over a time of approximately 2 years with hourly sampling, and with information about contamination, throughput, chemical analyses and control parameters. Goal of the analysis is to find the (causal) dependency structure leading to the final processing outcome quality.

This is done using a combination of graphical lasso [4] and Granger causality [1] methods. The former detects correlations between variables which cannot be explained by indirect influences via intermediate variables. The seconds one makes use of the fact, that causes usually occur (and should be detected) before their dependent effects, to establish a causal direction in the found correlations.

Parallelization of this approach is needed because of cubic dependence of problem size on number of features[1] and number of time lags considered in the Granger Causality detection, resulting in a problem size too large (wrt. memory and computaton time) for a common single system. At the lowest level, the use of linear algebra methods implies according potential for either employment of data parallelization, and/or the use of highly optimized existing numerical libraries. On a higher level, the linear algebra operations are utilized inside loops, which allow parallelization using stream parallel processing. As an example, coordinate descent, used for parameter optimization, allows a certain (data dependent) degree of parallelization [3].

Considering this problem structure, the use case provides an interesting example for the study of parallelization using ParaPhrase and other frameworks. The highest level parallelism is constituted by independent tasks, and allows coarse grained parallelization. On an intermediate level, the loop in coordinate descent can be parallelized up to a data dependent degree, as studied in the next section. The lower level linear algebra is parallelizable in a rather fine grained data parallel way. This suggests that the problem would be well suited for a heterogeneous parallel architecture, with CPU multicores working on the coarse grained parallel part, supported by GPGPUs for the fine grained parallelism.

[1] That is for dense problems, ie. a high number of correlations between variables; for sparse problems, it can be considerably less.

## 3 Parallelization approach

In the ParaPhrase approach [6], the parallelization is developed as a graph of components forming certain patterns of parallel execution, with sequential code or sub-patterns inside the components. The components are statically mapped to the available computing resources (CPUs and GPGPUs), with optional dynamic remapping dependent on runtime performance and actual load of resources. Refactoring can be used for parallelizing sequential algorithms and for transforming sets of patterns into functionally equivalent sets, which might yield other (better) non-functional (e.g. performance) behaviour.

The ParaPhrase parallelization approach has the following nice properties, from a use case perspective, and as relevant to the development of *mlpp*:

- Synchronization issues are automatically handled by the pattern implementations. This way, race conditions and deadlocks are not a problem.
- Clear structure of code and communication. This results in code which is easily understandable by others, and reduces the probability of bugs.
- Flexibility with respect to target hardware. The methods developed are usually used in several projects and at several customer sites. The hardware available there is subject to variation, and the possibility to have the static and dynamic mapping of components adapt to the target system greatly enhances the possibility for code reuse.

The graphical lasso method used here as an example consists of the following parts inside the critical loops:

1. A loop over the independent unconnected components in the graph (having almost no correlation among any of the member variables)
2. An iteration until convergence of the following steps
3. A loop over each of the variables in the current component
4. *Coordinate descent* on the weights/correlations of this variable
   a. An iteration until convergence of the following steps
   b. A loop over each of the variables in the current component
   c. Linear algebra operations computing the weight for one variable, dependent on the other variables in the current component

In the last step, the weight update for one variable is dependent on the weights of the other (connected) variables. This introduces a coupling of the variable weight updates making the parallelization non trivial. One way [3] to paralllelize these steps is to use the weights of the last iteration, or of the current one if their potentially concurrent computation has already been completed. This can lead to slower convergence, trading off the quicker inner loop against more outer loop iterations. It can even lead to divergence. But in the cited paper, the authors present a proof that by limiting the parallelism degree to a data dependent number, the spectral radius of the correlation matrix, convergence can be guaranteed.

Thus, this use case gives the opportunity to study parallelization with respect to the following characteristics:

- The design and clarity of parallelization for nested loops/functions

- The implementation of non-trivial dependencies between components, and parallelization dependence on data characteristics
- The behavior of nested parallelized components

## 4 Comparison to OpenMP

An important point in evaluating the ParaPhrase parallelization approach is the comparison to other existing approaches. Based on our work on parallelizing the Coordinate Descent algorithm we have made the following experiences.

ParaPhrase in its current state of development allows parallelization for shared-memory systems. Therefore OpenMP has been chosen as the first technology for comparison as the core targets of OpenMP are shared-memory multicore systems, too. In the future ParaPhrase will also support distributed systems as well as GPGPU. OpenMP in contrary, though first implementations for distributed memory systems exist, has to be used in conjunction with MPI to be used on distributed systems. GPGPU is not supported.

Parallelization of Coordinate Descent using OpenMP already reveals several differences to the parallelization using the ParaPhrase approach.

Using ParaPhrase only needs linking of a library[2] and using the provided classes and methods in the included header files. Porting between different platforms, hardware as well as operating systems, is therefore quite easy. OpenMP on the other hand relies on compiler directives, library calls and environment variables. The used compiler therefore has to support OpenMP. The fact that not all compilers support OpenMP and that different compilers support different versions of the OpenMP standard makes porting and platform-independence more difficult.

A further major difference is the abstraction level. Whereas ParaPhrase follows an abstract pattern-based approach, OpenMP has a lower level view, mainly parallelization of loops (data-parallelization) and parallelization of sections (task-parallelization). The different approaches yield several implications.

Generally ParaPhrase, as a more abstract approach, requires more extensive changes of existing sequential code whereas OpenMP, mainly based on using directives, only needs minor changes and allows incremental parallelization. First parallelization prototypes only concentrating on parts with the highest potential for performance gain can be implemented quickly and easily. But as soon as parallelization requires more complicated constructs like synchronization or nested parallelization, OpenMP may also require greater code changes and may furthermore lead to confusing nested constructs, which are more error-prone. E.g. race conditions within parallelized loops may lead to almost no performance gain and need greater changes to avoid them.

Another consequence of the compiler directive based approach of OpenMP is that it is hard to determine what is really going on behind the scenes and how parallelization is carried out by the compiler. This makes debugging and

[2] pthreads in the case of parallelization for multicore machines

profiling, which is already a challenging task in multithreaded applications, even more difficult.

Based on the first experiences using those to approaches of parallelization it has become apparent that OpenMP is easy to use at the beginning for parallelization of simple algorithms without difficult synchronization issues and complex data structures. The more complex the parallelization becomes, the more confusing it might get and the greater the changes of the sequential code may have to be. Diving deeper into OpenMP it furthermore gets revealed that the correct usage has several pitfalls. Placing directives at wrong locations, even if allowed, can lead to performance losses. The requirement for more extensive code changes due to the more abstract pattern-based approach of ParaPhrase on the other hand might need more time at the beginning,[3] but has several advantages. The separation of code leads to programs that are easier to understand and maintain, even if the parallelization is more difficult. The parallelization is far more traceable and therefore easier to debug and profile. The prospective support for distributed systems and GPGPU makes the usage of ParaPhrase also more flexible.

## 5 Status of ParaPhrase and influence on the workflow

ParaPhrase is a project in progress, with about a third of its time having passed. So far, the technical (Skeleton) foundation for implementation of the patterns is available, in form of the SourceForge project *FastFlow*. This includes several of the very flexible general purpose patterns *Pipeline*, *Farm*, *Map/Reduce*. Further (general purpose and domain specific) patterns will become available with time, leading to the possibility of an easy parallelization of further parts of the *mlpp* library.

Refactoring for C++ is not yet available, and as a consequence, currently the parallelization and introduction of patterns into the code has to be done without its support. Later on in the project, parallelization and especially the evaluation of different parallelization variants will become much easier by employing refactoring. It will automate much of the tedious process of program rewriting, and additionally assure that a transformed algorithm is syntactically well formed and functionally equivalent to the original. The refactoring is also supposed to include non-functional behaviour (runtime, memory requirements, etc.) in suggestions for sensible ways of rewriting an algorithm.

Distributed processing is available in ParaPhrase in an initial version, and GPGPU support is work in progress. At the end of the project, *mlpp* will thus support a wide range of parallel hardware.

Mapping currently has to be done by the programmer; on multicore machines that implies selecting the right number of workers for the *Farm* pattern, in the distributed case it also entails placement of computational components on individual machines. Both are expected to be optimized ahead of execution

[3] This is expected to be less of an issue, or even to be reversed, later on with the availability of refactoring. See next section on the status of ParaPhrase.

by the static mapping component of ParaPhrase later on. In the end, runtime information will also allow to perform dynamic mapping of components, taking into account actual computational load induced by unpredictable variations in computations, and by the load due to other concurrently running programs.

Concluding, the vision of ParaPhrase includes the most important requirements relevant to a parallelization of a Machine Learning library like *mlpp*. On the other hand, the realization of this vision is quite ambitious, and there remains a lot of work to be done for making it come true.

## References

1. A. Arnold, Y. Liu, and N. Abe. Temporal causal modeling with graphical granger methods. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '07, page 66–75, New York, NY, USA, 2007. ACM.
2. R. Bekkerman, M. Bilenko, and J. Langford, editors. *Scaling Up Machine Learning - Parallel and Distributed Approaches*. Cambridge University Press, 2012.
3. J. K. Bradley, A. Kyrola, D. Bickson, and C. Guestrin. Parallel coordinate descent for L1-regularized loss minimization. In *Proceedings of the 28th International Conference on Machine Learning*, Bellevue, WA, USA, 2011.
4. J. Friedman, T. Hastie, and R. Tibshirani. Sparse inverse covariance estimation with the graphical lasso. *Biostatistics*, 9(3):432–441, July 2008.
5. J. Gonzalez, S. Singh, A. Zheng, G. Taylor, J. Bergstra, M. Bilenko, and Y. Low. NIPS2011 workshop on big learning. `http://biglearn.org/2011/index.php/Schedule`.
6. K. Hammond, M. Aldinucci, C. Brown, F. Cesarini, M. Danelutto, H. Gonzalez-Velez, P. Kilpatrick, R. Keller, M. Roßbory, and G. Shainer. The ParaPhrase Project: Parallel Patterns for Adaptive Heterogeneous Multicore Systems. In *FMCO 2012*, 2012.
7. S. Singh, J. Duchi, Y. Low, and J. Gonzalez. NIPS2012 workshop on big learning: Algorithms, systems, and tools. `http://biglearn.org/index.php/Schedule`.