Parallelization of Stochastic-Local-Search Algorithms using High-Level Parallel Patterns

Michael Rossbory
Software Competence Center Hagenberg
michael.rossbory@scch.at

Georgios Chasparis
Software Competence Center Hagenberg
georgios.chasparis@scch.at

Abstract

Mathematical models for optimization can help companies optimizing their production and planning processes and therefore to reduce costs and increase quality. But applying such models effectively is challenging. Developers need expertise in mathematics and skills in software development to implement them. Furthermore optimization algorithms are inherently computationally very intensive. Parallelization reduces this computation time severely, but adds additional complexity, especially when low-level parallelization techniques are applied. Therefore developers would have to be experts in concurrent programming, too.

In this paper we present a stochastic-local-search algorithm to solve such an optimization problem from industry encountered in the slitting of metal sheets used in the production of electrical transformers. Furthermore we introduce a high-level pattern based parallelization approach that has been developed in the *ParaPhrase* project, depict how it can easily be applied to parallelize this optimization algorithm, without introducing the additional complexity of traditional low-level approaches, and describe why and how parallelization improves the result of the optimization process.

Keywords parallel pattern, discrete optimization, paraphrase, stochastic local search algorithms

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 for in roll cutting in clothing [5] or paper industry [6], optimization of control loops in chemical plants or oil refineries or optimization in supply chain management or logistics, just to name a few. The case study in this paper 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. A detailed description of the case study will be given later in this paper.

The research reported in this paper has been partly supported by the Austrian Ministry for Transport, Innovation and Technology, the Federal Ministry of Science, Research and Economy, and the Province of Upper Austria in the frame of the COMET center SCCH.

Several methodologies have been developed to address cutting-stock problems or bin-packing problems, including linear-programming (LP) based approaches [8, 9] 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* [11] 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., [11]) need to be implemented, including a) *reprocessing candidate solutions* from earlier optimization stages, and b) *experimenting alternative processing paths.*

In this paper, we investigate the utility of parallelization in the execution of such diversification strategies. *However, parallelization adds additional complexity to the system and makes it harder to develop and maintain and more error-prone, especially when low-level parallelization techniques are used. Expertise in concurrent programming is therefore highly required.*

Applying a high-level pattern-based parallelization approach, as developed in the *ParaPhrase* project, reduces this additional complexity caused by parallelization to a minimum and reduces needed expertise in concurrent programming.

The remaining of the paper is structured as follows. After a description of the use case and the basic idea optimization algorithm, an overview about *ParaPhrase* technology with focus on high-level and domain-specific patterns will be given. After that the application of the pattern used for parallelization will be explained. An evaluation of the parallelization approach and the resulting performance will be given at the end of the paper.

2. Use Case Description & Objective

In this paper, we are particularly concerned with a generalized version of the so-called 1/V/V/R cutting-stock problem [7]. 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

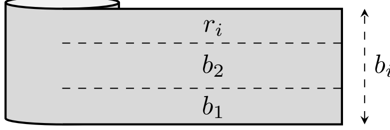

Figure 1. Roll slitting.

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].

2.1 Cutting-stock problem (background)

In this section, we briefly describe 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 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.,
$$b_j \sum_{i=1}^{m} x_{ij} \ell_i d_i \geq w_j, \tag{1}$$
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
$$r_i(X) \doteq b_i - \sum_{j=1}^{n} x_{ij} b_j, \tag{2}$$
should always be within a finite set $\mathcal{R}$ of allowable residual widths.

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:

$$\min_{X \in \mathbb{Z}_+^{m \times n}} \sum_{i \in \mathcal{I}} w_i \mathbb{I}_{\{\exists j \in \mathcal{J} : x_{ij} > 0\}}, \tag{3}$$

where

$$\mathbb{I}_A \doteq \begin{cases} 1 & \text{if } A = \texttt{true}, \\ 0 & \text{else.} \end{cases} \tag{4}$$

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

The objective function (3) subject to the job-admissibility constraint (1) and the rest-width-admissibility constraint (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 [7], or minimization of rolls needed by the assignment when the objects are identical, as considered by [9]).

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 [9]) 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 [11]). 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.

3. Optimization Algorithm

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 (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. [12, 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 (1) is satisfied. However, in most cases, satisfying even constraint (1) may be rather challenging, thus more sophisticated initialization algorithms may be required. An

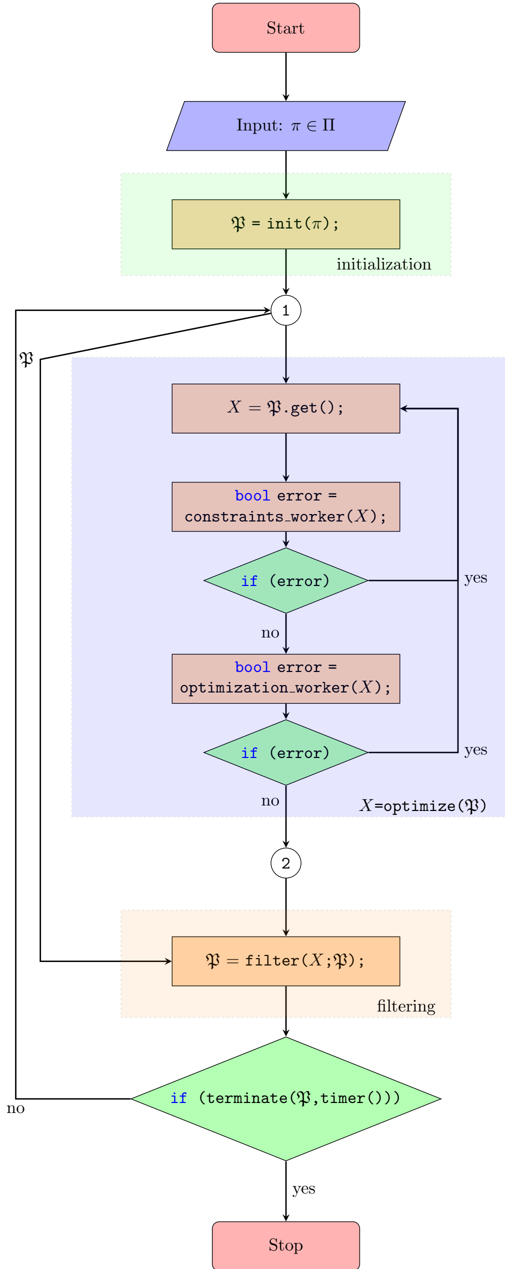

Figure 2. Architecture of optimization algorithm.

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 solutions which satisfy all imposed constraints, and b) candidate solutions which improve the cost of (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

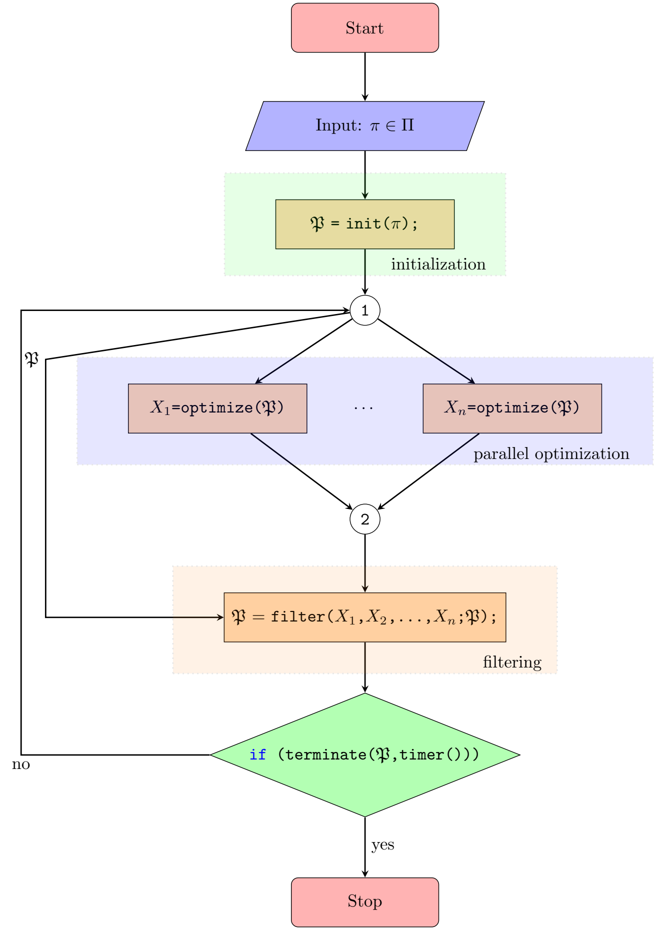

Figure 3. Architecture of parallelized optimization algorithm.

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., [11, 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., [11, 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.

Since the goal of this paper is to present a generalized framework that may accommodate stochastic-local-search algorithms for cutting-stock problems, the details of the functions `optimize(`$\mathfrak{P}$`)` and `filter(`X,$\mathfrak{P}$`)` go beyond the scope of this paper and its details will not be presented here.

4. Parallel Patterns

The *ParaPhrase* [17] project introduced a new way of parallelism in software development for heterogeneous architectures using

high-level parallel design patterns in conjunction with advanced refactoring technologies.

Parallel design patterns describe solutions of recurring problems in parallel programming. They are an abstract entity that provide no implementation that can be directly used in an application. Design patterns are independent from the actual software implementation and hardware architecture. Parallel design pattern are comparable to traditional design patterns in software development [13]. Patterns are designed to be simple and generic and it is not intended to provide a pattern for every parallelization problem that might occur. To achieve the desired parallel behavior patterns can be combined or nested to solve a particular problem.

The patterns itself are implemented in terms of skeletons, that might be higher order functions or template classes. In *ParaPhrase* these skeletons in turn are implemented in C++ based on the *FastFlow* [18, 19] library and in Erlang based on the *Skel* [20] library. They encapsulate all the low-level parallelization details like thread creation, communication or data access and hide them from the application programmer. Therefore the developer can focus on the design of the parallel behavior and just chooses the appropriate patterns or a combination of them.

The patterns introduced in *ParaPhrase* are classified into the following levels of abstraction [14]:

- Core patterns: basic building block of a parallel computation, that is a minimal set of quite simple as well as basic parallel exploitation patterns supporting composition and such that they can be used, alone or in composition, to describe a wide range of complex patterns. This set includes: pipe, farm, seq, map, and reduce.
- High-level patterns: general parallel patterns appearing in domain specific contexts and closer to the application programmers viewpoint and programming habit with respect to the core skeletons/patterns. This set includes: Divide and conquer, search, sort, pool evolution, and work-flow graph interpreter.

4.1 Pool Evolution Pattern

This section describes the pool evolution pattern in more detail, since this pattern is finally applied for parallelizing the optimization algorithm and is in use at our customers site.

The pool evolution pattern is inspired by the idea of evolutionary algorithms, generic population-based optimization algorithms, that in turn are based on mechanisms from biological evolution, like reproduction, mutation, or selection.

The workflow of the pool evolution pattern can be summarized as follows [14]:

A "candidate selection" function (s) selects a subset of objects belonging to an unstructured object pool (P). The selected objects are processed by means of a "evolution" function (e). The evolution function may produce any number of new/modified objects out of the input one. The set of objects computed by the evolution function on the selected object are filtered through a "filter" function (f) and eventually inserted into the object pool. At any insertion/extraction into/from the object pool a "termination" function (t) is evaluated onto the object pool, to determine whether the evolution process has to be stopped or it has to be iterated.

A pool evolution pattern therefore computes P as result of the following algorithm:

1: **while** $not(t(P))$ **do**
2: $\quad N = e(s(P))$
3: $\quad P = P \cup f(N, P)$
4: **end while**

Although initially designed to solve problems from the softcomputing application domain, especially the evolutionary computation field, due to its generic implementation the patterns can easily be used to solve problems from other domains [1]

5. Algorithm Parallelization

This section discusses the parallelization of the previously described optimization algorithm. To achieve a basic parallelization often more than one pattern is applicable. But not all suitable patterns give the same performance gain and are equal easy to deploy. Therefor we want to describe the possibilities parallelizing the optimization algorithm using different core as well as high-level patterns.

The basis for the parallelization was an already working sequential implementation of the whole optimization program including the algorithm described above. The main construct of the core optimization part as shown (Figure 3) is the 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 showed that this 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.

Within the optimize function the *contraints_worker* and *optimization_worker* would bear further potential for parallelization. But exploiting them too would lead to a huge number of threads running on the system and to achieve a great speedup also a huge number of cores would be needed, which is not the case for our deployment system. Therefore we focus on parallelization the main loop contruct only.

Different parallelization techonolgies have been investigated. The decision which one to use is based on several criteria.

- 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, since 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.

Obviously a simple for-loop parallelization, using e.g. OpenMP or the par-for pattern, is not possible, since the iteration space is not known in prior. The high-level pattern-based approach of *ParaPhrase* perfectly meets those criteria stated above. Possible core patterns that can be applied are *farm* and *pipe* and a combination of them. (Details about these patterns can be found in [15] [16] [14]). Furthermore the pool evolution pattern, as high level pattern, can be used.

5.1 Core Pattern Application

The following describes the application of different core patterns.

Pipe The *pipe* pattern fits to the basic idea of the optimization algorithm. Every step of optimization and constraint enforcing steps

to solve the optimization problem can be mapped directly to one stage in the pipeline pattern. The effort applying this pattern is equal to the *Farm* pattern. Applying the *parfor* pattern or *OpenMP* would be less effort, but this they do not provide the demanded flexibility. The existing code has to be changed in several places. Some classes have to derive from ff_node and additional classes have to be implemented that serve as emitter and collector. The degree of parallelization is bound to the number of steps in the optimization chain. E.g. if the chain consists of 10 steps 10 (+1 for the emitter thread) cores will be used even if more would be available. So at least 11 cores have to be available to achieve a good speedup. An advantage of the pipeline approach is that one optimization step is accessed only once at a time, since only one instance of each step exists.

Farm In the *farm* pattern the emitter of the farm takes the partial solutions from the pool and passes them to the workers. A worker wraps the loop that iterates over the all the optimization steps. In a first implementation also a collector has been used, to keep changes in the code as small as possible. The collector was responsible for calculating the objective value and pushing the processed solution back into the pool. In a second step the calculation of the objective value has been refactored so that it could be included into the worker of the farm and therefore calculated in parallel. Furthermore the collector could be omitted and replaced by the feedback channel to send the solutions back to the emitter. The emitter then decides whether to dismiss the solution, to process it further or to terminate the whole algorithm. Implementation effort of the first version is similar to the pipe approach. But refactoring of the implementation to be able to omit the collector to get a greater speedup increased the effort, but decreased execution time. Since there are shared resources whose access needs to be synchronized a linear speedup will not be possible.

Farm with nested pipes In this implementation the *farm* pattern and the *pipeline* pattern have been combined by nesting a pipe in each farm worker. This idea has the greatest potential for speedup, but even with the smallest reasonable number of farm workers, which is two, and an average length of the optimization chain of ten, 23 threads a running. So if the system does not have at least 23 cores (which was the case for our evaluation platform, it had 20 cores), there will be no speedup since the overhead for scheduling between the threads would be too high. Additionally, the higher the number of threads that have to be synchronized accessing the shared resources, the less efficient it is.

Common to all core patterns is that, although the effort is much less compared to traditional low-level approaches, the existing legacy code has to be changed on several places. Applying high-level patterns solved this problem for our use case.

5.2 High-Level Pattern Application

All of the above mentioned criteria are perfectly fulfilled by the pool-pattern, which was the pattern of choice at the end. The semantic behind the pool pattern fits perfectly to the basic idea of the optimization algorithm which is also based on a pool of possible solutions that are going to be optimized.

The effort of applying this pattern is limited to mapping the existing code to the functions for selection, evolution, filter and termination.

- The *selection* function simply returns all items of the pool, since in every iteration all candidate solutions are processed.
- The *evolution* function can be directly mapped to the optimize function.
- To *filter* the solution after the evolution phase the already existing filter function can be used.
- Also the implemented termination function can be directly mapped to the needed *terminate* function of the pattern.

The following code snippet illustrates the instantiation and use of the pool evolution pattern:

```
poolEvolution<shared_ptr< Workpiece<T> >, Env_t<T>>
    pool(_numWorker, _pool,
         selection, evolution, filter, termination,
         Env_t<T>(this));

pool.run_and_wait_end();
```

The template class of the pool pattern needs two template arguments for instantiation:

- Type of the items in the pool
- Type of an optional structure that holds additional data needed during pool execution

Constructor parameters:

- `_numWorker`: the number of workers used for execution the pool pattern; equal to the parallelization degree
- `_pool`: iterable structure that holds the pool items
- `selection, evolution, filter, termination`: pointers to functions
- `Env_t<T>(this)`: needed environment data needed during optimization process

The command *pool.run_and_wait_end()* synchronously executes the pool pattern and returns once the execution terminates.

In the original implementation of the pool evolution pattern only the evolution of the pool items is executed in parallel. That means for our particular case that the optimize function is applied on all selected candidate solutions simultaneously. Further loop parallelization within the optimize function is possible by simply nesting e.g. a par-for pattern. But as already stated earlier this is only meaningful on system with a very high number of cores. Additional speedup can be achieved by parallelizing the filter function. For example by nesting another pattern like par-for or pipe.

6. Evaluation

This section discusses the performance of the parallelized application concerning the performance of the parallelization and its effect on the result of the optimization.

Performance evaluation of the parallel version using the pool evolution pattern has been performed on a 64 bit Windows system (Windows 7) with two processor 6 cores each (dual-hexacore, Intel Xeon X5690, 3.47 GHz) with 24 GB of memory. HyperThreading support has been disabled, as usual in HPC.

To evaluate the performance two scenarios have been considered.

- How does the execution time of the optimization algorithm change with an increasing number of workers (constantly increasing the parallelization degree) while the result of the optimization is kept constant.

  Figure 4 shows a significant speedup and reduction of execution time with an increasing number of workers while the result of the optimization is always constant.
- How does the result of the optimization is improved when the execution time is kept constant and the number of workers is constantly increased.

Figure 5 shows the evolution of the transformer weight with increasing number of workers. For a medium sized transformer, as in this example, the weight reduces about 2 tons which reduces the overall costs significantly.

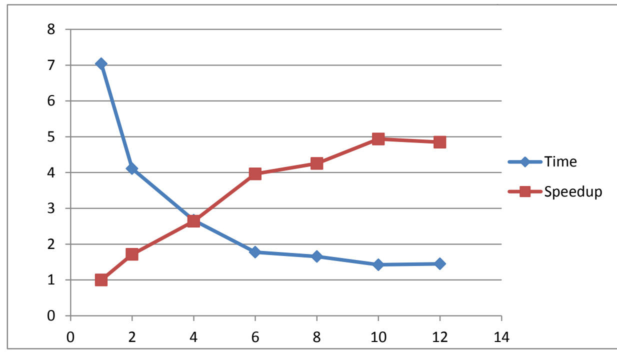

Figure 4. Speedup and Execution Time.

Figure 5. Weight decrease.

7. Conclusion

In this paper we addressed the parallelization of an optimization problem currently encountered in the slitting of metal sheets used in the production of electrical transformers. Due to the large number of constraints, convergence to local optima is rather common. This can be addressed by a larger number of initial admissible solutions and reprocessing from stages further away from the currently best solution. To countervail the resulting increase of processing time we decided to parallelize the optimization algorithms using high-level parallel patterns developed in the *ParaPhrase* project.

As already remarked compared to low-level parallelization approaches direct usage of the threading API, applying high-level methodologies has several advantages, like ability to focus on parallel behavior than on low-level details, better maintainability due to encapsulation of parallelization code within the parallel patterns, or increased portability between different operating systems.

Evaluation showed a significant decrease of execution time and improvement of the optimization process as direct result of parallelization. But furthermore we encountered that a parallelization degree higher than ten does not lead to further decrease of execution time (Figure 4). On the other hand using a parallelization degree of twelve still leads to a further weight decrease (Figure 5).

We are still investigating the reason for this phenomenon. One possibility is that the still sequential implementation of the filter function causes a bottleneck. Higher parallelization degree leads to a higher number of processed candidate solutions in an equal time period. Maybe the filter is not able to handle this increasing number and the candidate solutions queue up before the filter. Therefore we are currently working on a parallelization of the sequential implementation of the filter part to maybe solve the problem of stagnating speedup and to decrease the processing time even further. But even more important than the reduced processing time is the increase of the overall performance of the result of the optimization process.

References

[1] Aldinucci, Marco, et al. "Pool evolution: a domain specific parallel pattern." Proc. of the 7th Intl. Symposium on High-level Parallel Programming and Applications (HLPP), Amsterdam. 2014.

[2] Hammond, Kevin, et al. "The paraphrase project: Parallel patterns for adaptive heterogeneous multicore systems." Formal Methods for Components and Objects. Springer Berlin Heidelberg, 2013.

[3] Schoener, Holger and Robory, Michael "Patterns in Machine Learning: A new Parallelization Work-Flow for Machine Learning Methods", Proceedings of HLPGPU 2013, Berlin, Germany, 2013.

[4] Rossbory, Michael, and Werner Reisner. "Parallelization of algorithms for linear discrete optimization using paraphrase." Database and Expert Systems Applications (DEXA), 2013 24th International Workshop on. IEEE, 2013.

[5] Gradiar, Miro, Joe Jesenko, and Gortan Resinovi. "Optimization of roll cutting in clothing industry." Computers and Operations Research 24.10 (1997): 945-953.

[6] M.Helena Correia, Jose F. Oliveira, and J.Soeiro Ferreira. "Reel and sheet cutting at a paper mill." Computers and Operations Research, 31(8):1223 1243, 2004.

[7] Robert W. Haessler, Paul E. Sweeney."Cutting stock problems and solution procedures." European Journal of Operational Research 54, 141-150, 1991.

[8] C. Helmberg, "Cutting aluminium coils with high length variabilities," Annals of Operations Research, vol. 57, 1995, pp. 175-189.

[9] J.M. Valério de Carvalho."LP models for bin packing and cutting stock problems." European Journal of Operational Research 141, 253-273, 2002.

[10] A. Gerstl, S.E. Karisch, "Cost optimization for the slitting of core laminations for power transformers," *Annals of Operations Research*, vol. 69, 1997, pp. 157-169.

[11] H. Hoos, T. Stützle, "Stochastic Local Search: Foundations and Applications," Elsevier Inc., 2005.

[12] D.P. Williamson and D.B. Shmoys, "*The Design of Approximation Algorithms*," Cambridge University Press, 2011.

[13] Gamma, Erich, et al. "Design patterns: elements of reusable object-oriented software." Pearson Education, 1994.

[14] "Paraphrase project report. Final Pattern Definition Report (2013)". Available at http: //www.paraphrase-ict.eu

[15] "Homogeneous Implementation of Initial Generic Patterns (2012)". Available at http: //www.paraphrase-ict.eu

[16] "Final Homogeneous Implementation (2014)". Available at http: //www.paraphrase-ict.eu

[17] ParaPhrase web page (2014). http://www.paraphrase-ict.eu

[18] Aldinucci, Marco, et al. "Accelerating code on multi-cores with Fast-Flow." Euro-Par 2011 Parallel Processing. Springer Berlin Heidelberg, 2011. 170-181.

[19] FastFlow home page (2014). http://sourceforge.net/projects/mc-fastflow/

[20] Elliott, A., Brown, C., Danelutto, M., Hammond, K.: Skel: A Streaming Process-based Skeleton Library for Erlang (2012). 24th Symposium on Implementation and Application of Functional Languages, IFL 2012, Oxford, UK