Leibniz Universität Hannover
Fakultät für Mathematik und Physik
Institut für Algebraische Geometrie

in Kooperation mit
Technische Universität Kaiserslautern
Fraunhofer-Institut für Techno- und
Wirtschaftsmathematik

Computation of the GIT-fan using a massively parallel implementation

Masterarbeit

Hannover, 11.01.2018
Christian Reinbold
Matrikelnr.: 2942360

Erstprüfer:	apl. Prof. Dr. Anne Frühbis-Krüger
Zweitprüfer:	Dr. Janko Böhm
Betreuer:	apl. Prof. Dr. Anne Frühbis-Krüger
	Dr. Janko Böhm
	Dr. Mirko Rahn

Abstract

Fancy, english text.

Kurzzusammenfassung

Brillierender, deutscher Text.

Acknowledgements

Syn, Ack...

Contents

1 Introduction

Well, skip this and just read the rest. Less work for me.

2 Preliminaries

2.1 algebraic group actions

Torus as special case

2.2 Categorical quotients

2.3 GIT quotients & fans

Reduction of arbitrary groups to the torus case

2.4 Mori dream spaces, movable divisor classes?

3 Concept of the algorithm

3.1 Computing orbit cones

Monomial containment test Moving Cone

3.2 Traversing the GIT fan

(Theorie aus Kapitel 4 übernehmen)

Algorithm 1: Computing all neighbours of an GIT cone orbit

Input: a GIT cone orbit $\mathcal{O} \in {}^{\Lambda(\mathfrak{a},Q)(k)}/_{\mathcal{S}}$

Output: All GIT cone orbits $\mathcal{O}_{\mathcal{N}} \in {}^{\Lambda(\mathfrak{a},Q)(k)}/_{\mathcal{S}}$ such that $\{\mathcal{O},\mathcal{O}_{\mathcal{N}}\} \in E_{\mathcal{S}}$

$\mathcal{N} \leftarrow \emptyset$;
Choose λ such that $\mathcal{O} = \mathcal{S}\lambda$;
for μ such that $\{\lambda,\mu\} \in E_{\langle e \rangle}$ **do**
    $\mathcal{N} \leftarrow \mathcal{N} \cup \{\mathcal{S}\mu\}$;
end
return $\mathcal{N}$;

3.3 Exploiting symmetry

4 Implementation

In the following we present the implementation of the algorithm discussed in the previous chapter. In contrast to the existing Singular implementation gitfan.lib [3], which forms the foundation of our work, the approach described here utilises high performance computing methods in order to speed up the execution by running the algorithm on any number of machines simultaneously. Naturally, one has to identify independent components permitting concurrent execution on the one hand and inherent sequential processes on the other hand. As the sequential components of the algorithm mostly agree with gitfan.lib, we discuss only reworked parts and skip the remaining ones, referring the interested reader to [3]. The chapter is structured as follows: First, we discuss the employed parallelisation framework GPI-Space developed by the *Fraunhofer-Institut für Techno- und Wirtschaftsmathematik* (Fraunhofer ITWM) and follow up with the integration of singular code into GPI-Space applications. After encoding GIT cone orbits, we present a parallel design of algorithm 3 that is applicable for any kind of fan traversals. Finally, we describe input and output formats, additional features such as speeding up the execution by precomputed results, and executed software tests in order to verify the functional correctness of the program.

4.1 GPI-Space

GPI-Space is a workflow management system developed by the Fraunhofer ITWM which supports the execution of arbitrary workflows on ultra scale systems [6]. It consists of three key components:

- the distributed run-time engine (DRTS), which is responsible for building and managing arbitrary worker topologies based on the available compute nodes, resource management and job scheduling. It supports the reallocation of jobs in case of hardware failures and further integrates dynamically added hardware
- the workflow engine (WE). It tracks the state of the workflow, identifies a front of activities for which all dependencies are resolved and laces jobs

from input data and active transitions which then are sent to the DRTS for scheduling.

- a virtual memory layer, which provides the necessary infrastructure for the partitioned global address space (PGAS) programming model. It relies on GPI by Fraunhofer ITWM [11].

The framework has been developed with separation of concerns in mind, whereby the concerns here are given by computation and coordination [5]. In our case the computation takes place in sequential Singular-routines. The dependencies and data transfers between this routines, which assemble the particular routines into a complex, parallel algorithm, are described by a special domain language in the the coordination layer. In GPI-Space, this domain language is chosen to be an extension of the well known Petri net model introduced in 1962 by Carl Adam Petri [9]. Its formal nature permits a vast range of analysis and verification techniques. Concurrency is easily described due to locality of states. Furthermore, Petri nets yield precise execution semantics, supporting interruptions and restarts by differentiating between activation and execution of functional units. For this reason, fault tolerance to hardware failures is achieved by simply restarting failed executions. [1]

GPI-Space is shipped with an appropriate compiler that generates all files which are necessary for execution from a XML description of a given Petri net. Furthermore, GPI-Space also provides a basic visualization tool that comes in handy when debugging small to medium sized nets.

Petri nets

Formally, an (unweighted) *Petri net* is a triple (P, T, F) of *places* P, *transitions* T and directed *arcs* $F \subseteq (P \times T) \cup (T \times P)$ relating the former concepts. We demand that $P \cap T = \emptyset$. A function $M\colon P \to \mathbb{N}$ is called *marking* and describes a possible state of the Petri net. If we have $M(p) = k$ for a place p and the current state of the Petri net is given by M, we say that p holds k *tokens*. Visually, circles depict places, rectangles are transitions and arrows between circles and rectangles represent arcs whose direction is indicated by the tip. The current state M of the Petri net is displayed by placing $M(p)$ dots in the circle representing the place p. Figure 1 gives an example for the graphical representation of a Petri net.

A transition t is said to be *active*, iff

$$\forall p \in P\colon (p,t) \in F \Rightarrow M(p) > 0$$

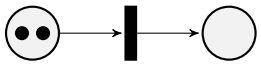

Figure 1: Graphical representation of a Petri net with two places and one transition. The current state is a marking that sends the left place to 2 and the second place to 0.

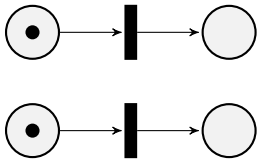

Figure 2: Modelling concurrent processes in a Petri net.

holds. In this case *firing* t means to transform the current state M into M' by the update rule

$$M'(p) := \begin{cases} M(p) - 1 & (p,t) \in F,\ (t,p) \notin F \\ M(p) + 1 & (p,t) \notin F,\ (t,p) \in F \\ M(p) & \text{else} \end{cases}$$

for all $p \in P$. We interpret this as follows: For every incoming arc (p,t) from p a token in p is consumed and for every outgoing arc (t,p') to p' a new token is placed into p'. Note that the notion of tokens "moving through transitions" instead of consumption and creation of tokens is erroneous as we will see later on when attaching data to them.

An important property of Petri nets is given by the fact that in many cases the same result is obtained if the order in which two transitions t_1 and t_2 fire is reversed. This allows the modelling of concurrent behaviour. When executing the Petri net model on a machine, the necessary steps for firing t_1 and t_2 respectively may be performed in parallel. Figure 2 depicts such a situation. Note that figure 1 also shows an example for concurrent behaviour in which the transition may fire twice. However, the order in which the tokens in the left place are consumed is of no relevance. This example seems trivial as tokens located at the same place are indistinguishable. However, this is not the case anymore when attaching data to tokens. In fact, figure 1 depicts the common scenario of concurrent behaviour in our algorithm, as parallel execution is mostly achieved by invoking the same routine for varying, independent input data.

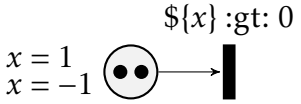

Figure 3: Guards in a Petri net. Only the token with $x = 1$ activates the transition.

Coloured Petri nets and guards

Workflows describe a (possible concurrent) progression of processes, hence it is evident to model a process by a transition. The input and output data of a process is taken into account by attaching data values of a predefined type to each token. Then every incoming arc provides input data for the process corresponding to the target transition, whereas every outgoing arc describes output data of the source transition that should be put into the target place. A place may only hold tokens of a single type, introducing strong typing to Petri nets. This kind of Petri net is called *coloured Petri net*, developed by Kurt Jensen in 1980. The term originates from the analogy of understanding each distinct type as a unique color. A formal treatment of this concept may be found in [8].

In order to control the data flow, each transition may be tagged with a predicate over the types of places with incoming arcs, which is called *condition* or *guard* [8]. The activation rule for a transition is extended such that not only a token has to be present for every incoming arc, but also the data values of the consumed tokens have to satisfy the condition. GPI-Space provides its own expression language in order to specify conditions. Figure 3 shows a transition that consumes tokens with positive data values only.

Expressions

GPI-Space supports two kinds of transitions. The first kind triggers a module call whenever firing, executing arbitrary C++code where the input and output tokens are mapped to references. The module call is packaged as a job and scheduled by the DRTS. A lightweight approach that requires no scheduling is given by the second kind of transition. The expression language provided by GPI-Space allows to specify the output data values of a transition by means of simple operations on the input data values. This extension is intended to be used for minor computations and saves the overhead of generating and scheduling a job by executing the expression directly within the WE. For instance, counting consumed tokens may be realised that way, see figure 4. If the counting would be

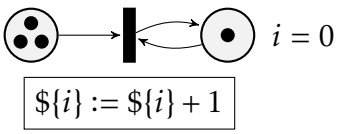

Figure 4: Expressions in a Petri net. After firing the transition three times, the left place contains no tokens anymore and the right place contains one token with $i = 3$.

realised with C++, each increment will cause a disproportionate communication and scheduling overhead.

Executing Petri nets

The WE is responsible for executing a given Petri net. It manages its current state and identifies active transitions. Whenever such a transition is detected, its input tokens are consumed, bundling the attached data values into a job that describes the execution of the transition. Then, this job is submitted to the DRTS where it awaits its execution. The set of active transitions not being executed already is called front of activities and generally contains numerous elements at once due to the locality of Petri nets. Thus, the DRTS is able to utilise the available capacities by scheduling multiple jobs in parallel.

When a job finishes, the results are returned to the WE which extracts the output tokens and updates the state of the Petri net accordingly. Note that during the whole process described above, the input tokens for an active transition are consumed whereas the output tokens are not available yet. In order to fully describe the state of an executable Petri net, one has to incorporate the knowledge about all active transitions and its cached tokens into the state description. Hence, the WE has to manage *timed*, coloured Petri nets. Figure 5 illustrates the process of executing two active transitions in parallel. The module called by the transition sleeps for x seconds, where x is the input data value, and then returns the increment $x + 1$.

The WE itself is executed on a single node of the underlying system. Hence, special care must be taken when designing Petri nets. Instead of firing numerous transitions with execution times of milliseconds, one is advised to aggregate these transitions into a single one with an execution time of several seconds. Otherwise, the WE quickly becomes a bottleneck since the amount of scheduled transitions per second does not scale with the number of available computing nodes. Furthermore, the marking of the Petri net is stored in the RAM of a single node. If large quantities of data have to be managed, the tokens should only

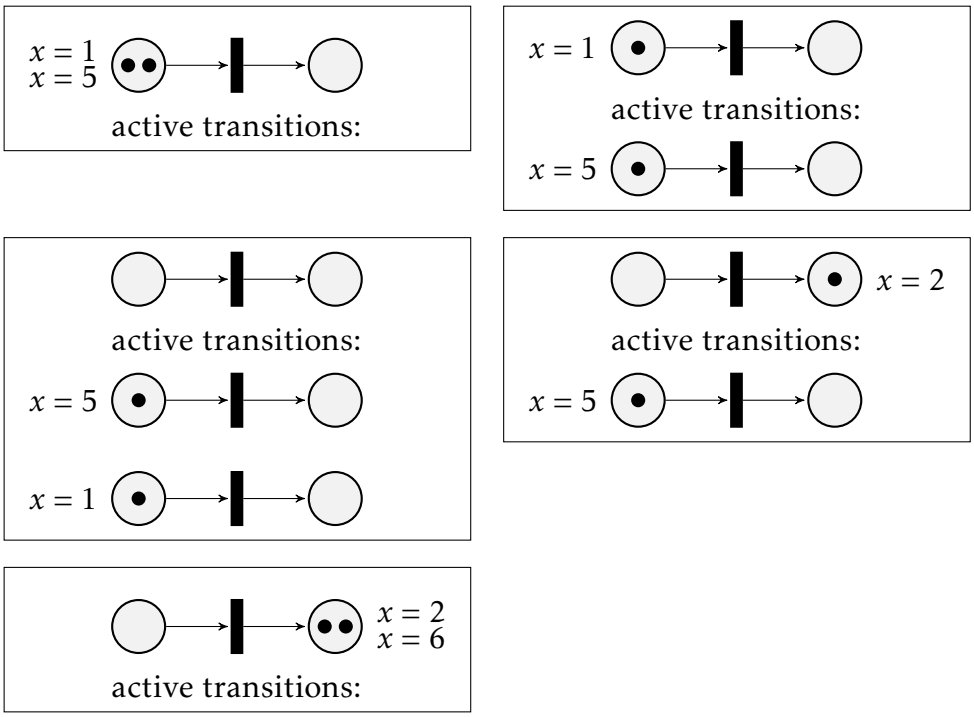

Figure 5: Possible execution of a Petri net. The module called by the transition sleeps for x seconds and then returns the increment $x + 1$.

contain a reasonable amount of metadata pointing to the real data which – for instance – is stored in a shared directory.

GPI-Space assumes that module calls have no side effects. Thus, when implementing the execution code for the transitions, possible racing conditions and concurrency issues have to be considered properly. Furthermore, each worker managed by the DRTS is executed in its own process, but beyond that, jobs assigned to the same worker run in the same environment. For this reason jobs are influenced by changes of global state by prior executions. This behaviour causes several issues when integrating Singular, which is implemented as a large, global state machine.

4.2 Integration of Singular

Singular is a computer algebra system for polynomial computations, with special emphasis on commutative and non-commutative, algebraic geometry and singularity theory [10]. Typically, algorithms utilising Singular are written in its own C-like programming language, which is interpreted at runtime. Besides providing a console frontend for code execution, Singular also comes with the C

library libSingular that implements its full backend. In particular, C methods for interpreting and executing Singular code are available. libSingular also exposes the internal data structures used by Singular, so that one can write conversion methods between Singular types and user defined types, i.e. GPI-Space types.

Since module calls in GPI-Space permit the execution of arbitrary C++ code, every legacy application with a C interface such as Singular can easily be integrated and executed. In order to execute Singular code in a module call, one has to perform the following steps:

1. Check if a Singular instance is already running on the current worker. If not, initialise Singular.
2. Convert GPI-Space types to Singular types.
3. Map the converted data to identifiers in order to address it by Singular code.
4. Formulate a C string containing the Singular code to be executed and pass it to the Singular interpreter via libSingular. Note that instead of hard coding large blocks of code, it is reasonable to outsource them into a Singular library. Then, this step reduces to loading the library and invoking a single method.
5. Extract the data mapped to all identifiers that contain results of the computation.
6. Convert the extracted Singular types to GPI-Space types that can be attached to tokens.

Drawback: Global state

Although integrating Singular code is sufficiently easy, the approach described above suffers from a serious issue that originates from Singular heavily relying on global state. For instance, loaded libraries, declared variables and the current basering are managed in global data structures. If module calls are supposed to have no side effects, all changes concerning global state have to be reverted before returning from them. Unfortunately, libSingular lacks the feature of creating and restoring images of its global state. In fact, there seems to be no way of resetting Singular to its initial state short of killing the current process, which – obviously – is not desirable.

Not being able to reset the Singular instance also makes writing decoupled test cases more problematic. In order to avoid tests being influenced by one another due to global Singular state, each test has to be executed in a separate process.

Unfortunately, this prevents the utilisation of many Google Test features such as bundling test cases, reusing data configurations and parametrising tests.

Drawback: No in memory serialization of Singular types

When GPI-Space has to share Singular data between module calls without knowing the internal structure of the data, this can be achieved by serialising the data into Binary Large Objects (BLOBs) which then are managed by the WE. Singular supports the serialization of its common data types by using *ssi file links*. As the name suggests, the serialization string always is written to a file. It would be desirable to also obtain a serialization string in memory without the overhead of using virtual files or reading the created ssi files. Currently, we pass metadata (that is file paths) pointing to ssi files in a shared directory instead of acquiring and passing BLOBs.

Drawback: Singular programming language lacking data structure implementations

Since the purpose of developing a parallel implementation of algorithm 3 is the computation of large examples within a reasonable timeframe, it is crucial to choose optimal data structures for large sets of data. Unfortunately, the Singular programming language lacks tree-like data structures with $\mathcal{O}(n)$ characteristics and, more importantly, hash tables with average time complexity of $\mathcal{O}(1)$ for insertion, searching and deletion. For this reason, all tasks dealing with the detection of duplicates, were moved from Singular into the C++ environment, where the Standard Template Library (STL) provides all necessary data structure implementations. Hence, the set of found GIT cones is managed in C++. The elimination of duplicates in orbit computations as well as the computation of the symmetry group action on GIT cones have been reworked in C++, resulting in a substantial performance gain compared to the gitfan.lib implementation.

4.3 Encoding GIT cone orbits

In chapter 3, we suggested to traverse a graph that reflects the structure of the maximal cones located in the GIT fan. Since nodes of the graph may be reachable by multiple paths, the traversal algorithm has to manage a list of found nodes such that no node is expanded twice. For this reason it is crucial to develop an encoding of nodes, i.e. GIT cone orbits, that

(i) conserves memory, permitting fast bit-per-bit comparisons,

(ii) allows to recover the encoded object from the code word,

(iii) does not require additional effort when computing code words and

(iv) is easily accessible for operations that are frequently performed on the nodes.

Encoding GIT cones

In order to encode GIT cone orbits, we require an appropriate encoding for GIT cones that allows us to apply the group action of $\mathcal{S}$ with minimal costs. The encoding described here originates from [4, Construction 4.3] and has been implemented in GITFAN.LIB. It utilises the result of (orbits mit p enthalten git cone), stating that every full dimensional GIT cone is the finite intersection of all full dimensional orbit cones containing the GIT cone. Thus, every GIT cone λ can be uniquely identified by bits indexed over $\Omega_{\mathfrak{a}}^{(k)}$ such that the bit indexed by $\vartheta \in \Omega_{\mathfrak{a}}^{(k)}$ is set to 1 iff $\lambda \subseteq \vartheta$. Formally:

$$\operatorname{code}\colon \Lambda(\mathfrak{a}, Q)(k) \to \{0,1\}^{\Omega_{\mathfrak{a}}^{(k)}}, \quad \lambda \mapsto (b_\vartheta)_{\vartheta \in \Omega_{\mathfrak{a}}^{(k)}} \quad \text{with} \quad b_\vartheta = \begin{cases} 1, & \vartheta \subseteq \lambda \\ 0, & \vartheta \not\subseteq \lambda \end{cases}.$$

Obviously, this mapping is injective and yields an encoding with code words of length $|\Omega_{\mathfrak{a}}^{(k)}|$, guaranteeing property (i). The GIT cone is recovered by intersecting all orbit cones indexing a 1 in the code word. Hence, (ii) holds.

Next, we verify property (iii). Since every GIT cone $\lambda(p)$ occurring in the algorithm is constructed by taking the relative inner point p and intersecting all orbit cones containing p, the code word can be computed alongside the construction of $\lambda(p)$ due to (orbits mit p enthalten git cone).

Finally, verifying (iv), we have to check that the group action of $\mathcal{S}$ on the code words defined by

$$\sigma \cdot \operatorname{code}(\lambda) := \operatorname{code}(\sigma \cdot \lambda) = \operatorname{code}(A_\sigma \cdot \lambda)$$

is realised by a low-cost operation on bit strings. Since we have

$$\lambda \subseteq \vartheta \;\Leftrightarrow\; A_\sigma \cdot \lambda \subseteq A_\sigma \cdot \vartheta,$$

the group action simply permutes the characters of code words in the same way as $\mathcal{S}$ acts on $\Omega_{\mathfrak{a}}^{(k)}$, see (Referenz: S wirkt auf OrbitCones). We are going to compute the group action $\mathcal{S} \circlearrowright \Omega_{\mathfrak{a}}^{(k)}$ in the next section, so that it is available during the traversal process. Hence, applying a group element to a code word is implemented as the application of a permutation which is stored in a lookup table.

Extending the encoding to orbits

After encoding GIT cones, we are able to encode GIT cone orbits by mapping each orbit to the code word of an uniquely identified representative. We choose the representative with the minimal code word regarding the lexicographical order over the alphabet $\{0,1\}$. Hence, the encoding is given by

$$\text{code}\colon \Lambda(\mathfrak{a},Q)(k)/_{\mathcal{S}} \to \{0,1\}^{\Omega_{\mathfrak{a}}^{(k)}}, \quad \mathcal{O} \mapsto \min_{\leq_{\text{lex}}} \{\text{code}(\lambda) \mid \lambda \in \mathcal{O}\}$$

and clearly is injective. Property (i) is inherited from the GIT cone encoding. The same applies to property (ii), since we already presented an approach for applying group elements to code words, so that we are able to recover the orbit from a single representative. Property (iii) holds since orbits typically are constructed from a representative and finding the minimal representative is cheap due to property (iv) of the GIT cone encoding. Finally, since operations on orbits usually are given by well defined operations on their representatives, (iv) also is inherited from the GIT cone encoding. One simply applies the operation on the GIT cone that is represented by the code word of the orbit.

4.4 Application flow & Concurrency

We differentiate between three stages when implementing the algorithm described in chapter 3. During the first stage, preprocessing of the input data and the identification of all orbits of $\mathfrak{a}$-faces with full dimensional image takes place. The second stage covers the computation of the orbit cones and the symmetry group action on GIT cone-hashes. This data is mandatory when determining the GIT fan in the third stage by traversing all maximal cones of the fan. Figure 6 depicts the simplified data flow of the algorithm in which each stage is collapsed into a single transition. Dashed arcs indicate read-only connections, that is data values of tokens in the source place are passed to the target transition without consuming them when it fires. Hence, *read-only* connections give access to static data computed once (such as processed input data), without discarding it due to consumption.

We reuse the notation from chapter 3. Recall that the matrix $Q \in \mathbb{K}^{k\times r}$ of rank r with columns $q_1,\dots,q_r$ encodes a torus action on the affine variety $X = V(\mathfrak{a}) \subseteq \mathbb{K}^r$ where $\mathfrak{a} \subseteq \mathbb{K}[x_1,\dots,x_r]$ is a homogeneous ideal with respect to the grading $\deg(x_i) = q_i$, $1 \leq i \leq r$. The positive orthant $\mathbb{Q}^r_{\geq 0}$ is denoted by γ. Furthermore, $\mathcal{S}$ (possibly trivial) denotes a symmetry group of the action of Q on X. (Notation mit Kap. 3 prüfen)

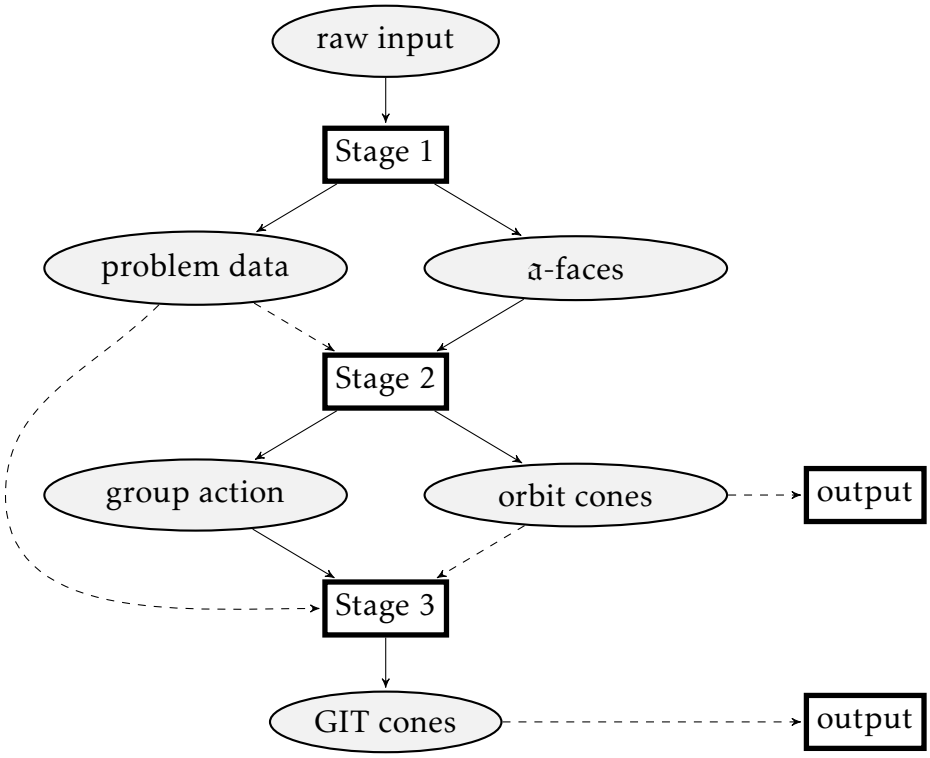

Figure 6: The data flow of algorithm 3 modelled as a Petri net with collapsed stages.

Stage 1: Preprocessing, $\mathfrak{a}$-faces

Stage 1 starts with preprocessing the raw input supplied by the user. During the process, we compute the following objects sequentially:

- The amount r of variables occurring in the basering and the dimension k of the ambient space in which the GIT fan lives. Both values are extracted from the matrix describing the torus action.
- the moving cone (optionally, see section 4.7)
- orbit representatives for the faces of the simplex γ under the symmetry group $\mathcal{S}$ in case it is nontrivial and no set of precomputed orbit representatives is provided (see section 4.7).

In a next step, the set of orbit representatives for the faces of the simplex γ is partitioned into parts of equal size. A token is generated for every part, which then is processed in parallel, selecting only the $\mathfrak{a}$-faces with full dimensional image under Q. Finally, the reduced parts are merged into a single token that then is passed to the next stage. The Petri net modelling this process is depicted in figure 7.

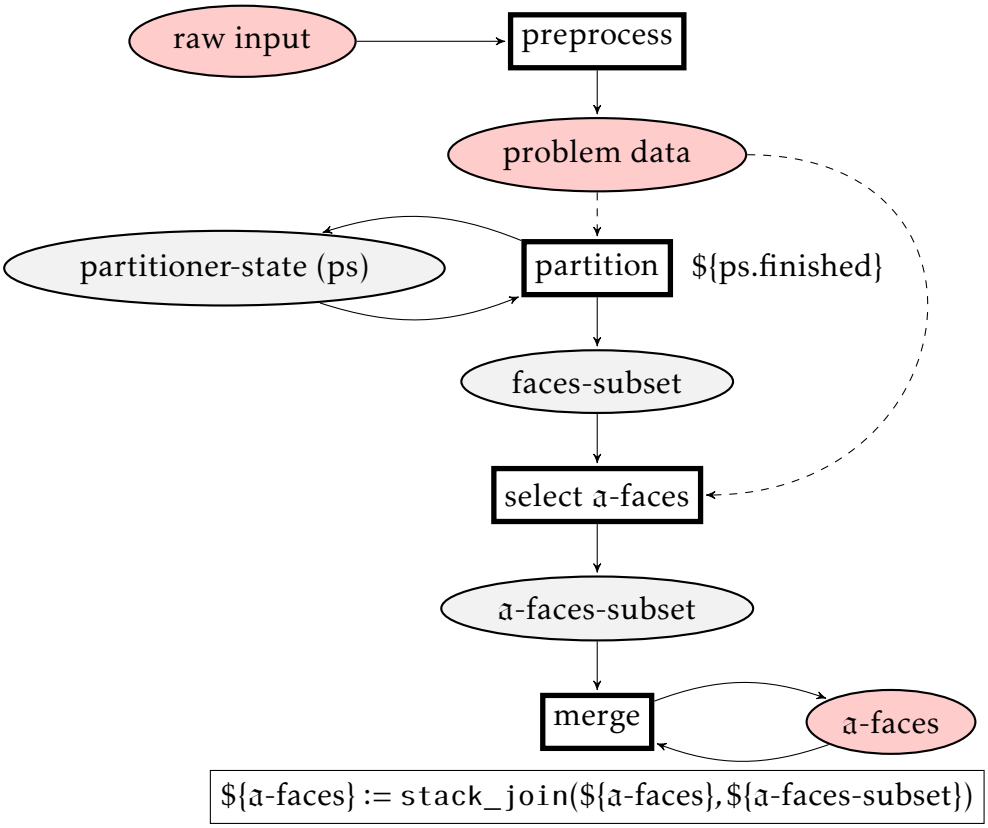

Figure 7: Stage 1 of the algorithm. After preprocessing the input and partitioning all faces, each part is reduced and merged into a single token. Highlighted places correspond to places in figure 6.

Since the [partition]-transition has to modify a state token which describes the faces assigned to a part already, it may execute sequentially only. This behaviour is enforced by adding the [partitioner-state]-place with a token describing the initial state of the partitioning process. The [𝔞-faces]-place is initialised with a token containing an empty list. Since the [merge]-transition is synchronised via this token, it also may execute sequentially only. However, the computationally heavy task of identifying 𝔞-faces is represented by the [select 𝔞-faces]-transition, which is not synchronised. Therefore, any amount of faces may be processed simultaneously.

Stage 2: Orbit cones, group action on GIT cone-hashes

After identifying a representative system for the 𝔞-faces with full dimensional image, we are able to determine all full dimensional orbit cones by taking the

image under Q and computing orbits under the symmetry group $\mathcal{S}$ acting on $Q(\gamma)$. The following steps are performed in successive order:

1. Compute the cones $Q(\gamma_0)$ for all $\mathfrak{a}$-faces γ_0 returned by stage 1. These images contain a representative system for the set of full dimensional orbit cones $\Omega_{\mathfrak{a}}^{(k)}$. (Notation bzgl. Kapitel 3 anpassen + Verweis auf Beweis der Aussage)
2. Optionally (see section 4.7): Intersect all obtained cones with the moving cone. Since the moving cone is invariant under the symmetry group $\mathcal{S}$ (Referenz), the resulting cones contain a representative system for $\Omega_{\mathfrak{a}}^{(k)}$ intersected with the moving cone.
3. For each obtained cone σ: Compute the finite orbit $\mathcal{S}\sigma$ and the group action of $\mathcal{S}$ on $\mathcal{S}\sigma$.
4. Eliminate duplicate orbits so that the remaining orbits form a partition of $\Omega_{\mathfrak{a}}^{(k)}$.
5. Aggregate all orbits into a single list containing $\Omega_{\mathfrak{a}}^{(k)}$. Utilise the previously determined group actions on the orbits in order to construct the group action of $\mathcal{S}$ on $\Omega_{\mathfrak{a}}^{(k)}$.

With the exception of step 5, in all steps a prescribed operation has to be executed on each element of a list of either cones or orbits. Hence, we are able to parallelise the computations in each step by partitioning the list into equally sized parts as described for stage 1. Note that the elimination of duplicates in step 4 translates to selecting all list entries such that no subsequent list entry equals the selected one.

The most computationally intensive operation is performed in step 3. Every group element is applied to the representative. Since the stabiliser subgroup may be nontrivial, we have to take care of possible duplicates. In order to do so, whenever a new cone is determined, the GITFAN.LIB implementation runs expensive comparisons with all previously found orbit elements and discards the cone if any matches are detected. We reworked the approach by exchanging the expensive comparison of cones with a string comparison of the cone's serialization, which is one to one for cones returned from SINGULAR's `canonicalizeCone` routine. Furthermore, we manage the found cones in a hash table (Referenz zu einer Hashtable-Einführung?), using a hash function on the serialization string. Since the maximal orbit size is known to be the cardinality of $\mathcal{S}$, we can adjust the size of the hash table such that hash collisions are unlikely. Hence, the duplicate elimination comes for free by performing only one string comparison on average.

The GITFAN.LIB implementation also compares cones in order to determine the action of $\mathcal{S}$ on the git cone orbits. However, the action may be constructed from

the cayley table of $\mathcal{S}$ as long it is known which group elements map to the same orbit element. For this reason we drop the cone comparisons and compute the action by composing permutations in $\mathcal{S}$, which is a very cheap operation scaling with the amount r of variables occurring in the basering..

Stage 3: GIT fan traversal

(Formalen Teil zum Algorithmus nach Kapitel 3 auslagern)

Now we are ready to determine the GIT fan. As described in chapter 3, this can be achieved by successively computing neighbours of full dimensional GIT cones. Note that the support of the GIT fan is convex, thus connected. From an algorithmic point of view, the GIT fan traversal translates to a graph traversal with the GIT cones being the nodes and the property of two GIT cones sharing a common facet being the edges. Formally, we consider the undirected, connected Graph $G_{\langle e \rangle} = (V_{\langle e \rangle}, E_{\langle e \rangle})$ with

$$V_{\langle e \rangle} := \Lambda(\mathfrak{a}, Q)(k),$$
$$E_{\langle e \rangle} := \{\{\lambda_1, \lambda_2\} \mid \lambda_1, \lambda_2 \in \Lambda(\mathfrak{a}, Q)(k) \text{ and } \lambda_1 \cap \lambda_2 \text{ is a facet of both } \lambda_1 \text{ and } \lambda_2\}.$$

Taking the symmetry group $\mathcal{S}$ into account, it suffices to traverse the graph $G_{\mathcal{S}} = (V_{\mathcal{S}}, E_{\mathcal{S}})$ with

$$V_{\mathcal{S}} := \Lambda(\mathfrak{a}, Q)(k)/_{\mathcal{S}},$$
$$E_{\mathcal{S}} := \left\{\{\mathcal{S}\lambda_1, \mathcal{S}\lambda_2\} \mid \{\lambda_1, \lambda_2\} \in E_{\langle e \rangle}\right\}.$$

Since $G_{\langle e \rangle}$ is connected, it immediately follows that $G_{\mathcal{S}}$ is connected as well.

(Ende formaler Teil)

In order to traverse any undirected, connected graph $G = (V, E)$, the following has to be provided:

- A computable encoding of V, that is an injective function mapping V to binary words.
- A routine `START` that computes an initial node $v_0 \in V$.
- A routine `NEIGHBOURS` computing the function $\mathcal{N} : V \to \mathcal{P}(V)$ which lists all neighbours of a node v, that is $f(v) = \{w \mid \{v, w\} \in E\}$.

In our case, we use the GIT cone orbit encoding described in section 4.3. `START` is implemented as in GITFAN.LIB, that is selecting random points $p \in Q(\gamma)$ until the corresponding GIT cone $\lambda(p)$ (Notation konsistent?) has full dimension. Then,

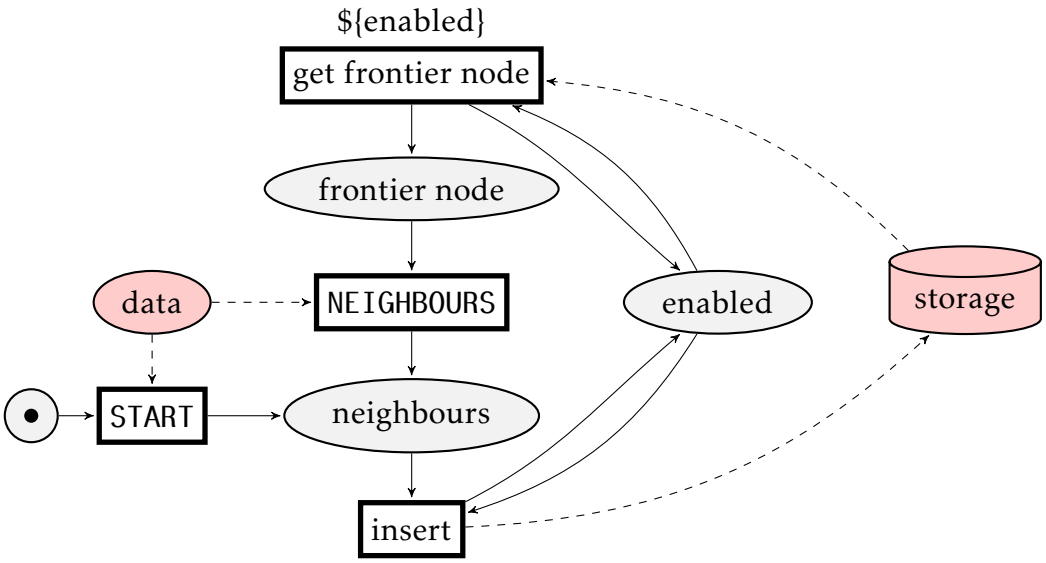

Figure 8: Modelling a graph traversal algorithm as a Petri net. Highlighted elements correspond to places in figure 6.

START returns the GIT cone orbit-code of $\mathcal{S}\lambda(p)$. The routine NEIGHBOURS is given by algorithm 1.

It remains to model a Petri net describing a parallel graph traversal. Note that this model is applicable for any kind of fan traversal, e.g. in the context of tropical varieties (Referenz von Janko), as long as the above mentioned demands are satisfied. Figure 8 depicts an appropriate Petri net together with a storage interface that allows to record visited nodes outside of the WE. The highlighted [data]-place combines the results of stage 2, which are required by the START and NEIGHBOURS routines for the GIT fan application.

We distinguish between two types of nodes. *Frontier nodes* are known by the traversal process. However, their neighbours have not been calculated yet. Hence, frontier nodes form the boundary of the ongoing graph traversal. The boundary is expanded by computing the neighbours of a frontier node, obtaining a set of frontier nodes. Nodes whose neighbours have been determined already are called *expanded nodes*.

The storage interface has to provide insertion and retrieval operations, which are used by the [get frontier node]- and [insert]-transition respectively. Whenever a node should be inserted, the storage implementation has to check if it is an unknown node, meaning that is does neither occur in the set of frontier nodes, nor in the set of expanded nodes. In this case the node is inserted into the set of frontier nodes. Otherwise, the node is discarded in order to avoid repeated

expansion of the same node. A boolean return value denotes whether an insertion took place. The retrieval operation returns a frontier node and relocates it to the set of expanded nodes. If the set of frontier nodes is empty, no node is returned.

The graph traversal is initiated by computing an initial node, executing START. The incoming place of START containing exactly one token ensures that the transition fires once only. The initial node is put into the [neighbours]-place so that it is published to the storage interface as the first frontier node when the [insert]-transition fires.

[get frontier node] frequently polls the storage interface for frontier nodes and generates tokens for each returned node. It is synchronised via the [enabled]-place containing a single token, preventing simultaneous access to the storage implementation that may not be thread safe. As soon as the storage interface does not return any frontier node, [get frontier node] disables itself by setting the value of the [enabled]-token to `false`.

The tokens representing frontier nodes are processed by computing adjacent nodes in [NEIGHBOURS]. Then, the results are inserted by invoking the storage interface. Note that the [insert]-transition also is synchronised via the [enabled]-place in order to prevent simultaneous access to the storage implementation. Whenever the storage interface indicates that at least one node has been inserted, the value of the [enabled]-token is set to `true`, reactivating [get frontier node].

The traversal is finished when no frontier nodes are available anymore and no nodes are processed currently. These conditions are met when the Petri net described in figure 8 deadlocks. In our implementation, we extended the Petri net by detecting such deadlocks and outputting all nodes, i.e. GIT cone orbits, then.

4.5 Storage implementations used in graph traversals

In the previous section we described a parallel graph traversal algorithm in order to compute the GIT fan. This algorithm depends on a storage interface, which is accessed synchronously. On that account, it poses a potential performance issue, slowing down the computation up to a point where all but one process are blocked due to storage access. For this reason it is of utmost importance that the storage implementations is able to handle hundreds of requests during a single call to NEIGHBOURS. Furthermore, it has to cope with arbitrary large sets of nodes. We pursue two approaches, storing the nodes either in a shared directory on disk or in memory on a single node.

4.5.1 Storing nodes on disk

Our first approach stores nodes by exploiting the scalable cluster file system BeeGFS, developed by the Fraunhofer ITWM [2]. Every stored node is represented by an empty file with a path that depends on the binary representation of the node. Thus, searching for nodes is realised by the system call `fstat`, whereas insertions are realised with `touch`. This implementation offers the following advantages:

- It is easy to implement since BeeGFS already is installed on the cluster system of the Fraunhofer ITWM. Mirroring the storage directory on every node of the cluster system is realised by BeeGFS and hidden from the application.
- Straightforward recovery in case of failures, e.g. power failures. The current state of the graph traversal is saved in persistent storage at any time.
- Practically infinite scaling due to BeeGFS.

However, load tests have shown that the response time by BeeGFS are far too high, taking up over 90% of the total computation time when computing the GIT fan of the $\overline{M}_{0,6}$ example on a moderate amount of 10 nodes with 16 cores each. For this reason we provide another, less scalable implementation with better response times that easily suffices the needs for computing the $\overline{M}_{0,6}$ example.

4.5.2 Storing nodes in memory

Instead of storing nodes on disk, we host an RPC-server on a single node of the cluster. This server implements the interface by managing the sets of frontier and expanded nodes in a hash table implementation provided by the STL. Consequently, all nodes sent to the RPC-Server are stored in the RAM of a single node. Since insert and search operations perform in $\mathcal{O}(1)$ as long as enough memory is available, this approach scales until a certain threshold is passed.

In the $\overline{M}_{0,6}$ example we have to store up to 249 604 entries of 106 bits each, summing up to about 3 MB of data. Thus, we are far away from reaching the limit of up-to-date RAM sizes. If one intends to run graph traversals consuming more memory than being available on a single node, the current storage solution may be extended by an implementation of distributed hash tables, utilising the RAM of every available node (Referenz).

When considering response timings, the second approach performs significantly better than the implementation using BeeGFS. Overhead induced by awaiting responses of the RPC-server is negligible even when executing the algorithm on 40 nodes with 16 cores each (see (Forward-Referenz auf Performance-Kapitel)).

4.6 Input & Output structure

4.7 Additional features

an Kommandozeilenparameter entlanghangeln

4.8 Software testing

5 Applications

5.1 Grassmanian $\mathbb{G}(2,5)$

5.2 Moduli space of stable curves of genus 0

6 Conclusion

Let $p = x_1x_3 + x_2x_4 + x_5 \in \mathbb{K}[x_1, x_2, x_3, x_4, x_5] =: \mathbb{K}[\mathbf{x}]$. Set

$$\mathfrak{a} := \langle x_3, x_4 \rangle \cdot \langle p \rangle,$$

$$Q = (q_1, q_2, q_3, q_4, q_5) := \begin{pmatrix} 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 2 \end{pmatrix}.$$

Note that Q defines a $\mathbb{Z}^3$ grading on $\mathbb{K}[x_1, x_2, x_3, x_4, x_5]/\mathfrak{a}$ since p and therefore $\mathfrak{a}$ is homogenous with respect to the grading $\deg(x_i) = q_i$. Thus, we obtain a torus action of $(\mathbb{K}^*)^3$ on $V(\mathfrak{a})$.

The $\mathfrak{a}$-faces with full dimensional image in $Q(\gamma)$, $\gamma = \mathbb{Q}^5_{\geq 0}$, are given by

$$\begin{aligned} \gamma_1 &:= \langle e_1, e_2, e_5 \rangle \\ \gamma_2 &:= \langle e_1, e_2, e_3, e_5 \rangle \\ \gamma_3 &:= \langle e_1, e_2, e_4, e_5 \rangle \\ \gamma_4 &:= \langle e_1, e_3, e_4, e_5 \rangle \\ \gamma_5 &:= \langle e_2, e_3, e_4, e_5 \rangle \\ \gamma_6 &:= \langle e_1, e_2, e_3, e_4 \rangle \\ \gamma_7 &:= \langle e_1, e_2, e_3, e_4, e_5 \rangle \end{aligned}$$

This is easily seen by sending all variables x_i with $e_i \notin \gamma_j$ in $\mathfrak{a}$ to zero and check that the new ideal does not contain any monomials. (details see below)

The orbit cones $Q(\gamma_j) \in \mathbb{Q}^3_{z_1,z_2,z_3}$ have the following form when intersecting them with the $\{z_3 = 1\}$-plain (vice versa, the orbit cones are recovered by expanding the polytopes in z_3-direction to cones in $\mathbb{Q}^3$):

A Something appendixable

Theorem A.1 (Great theorem) *Problem in appendix.*

Proof: by intimidation. □

Bibliography

[1] W. M. P. van der Aalst. 'Three Good Reasons for Using a Petri-Net-Based Workflow Management System'. In: *Information and Process Integration in Enterprises: Rethinking Documents*. Ed. by Toshiro Wakayama, Srikanth Kannapan, Chan Meng Khoong, Shamkant Navathe, and JoAnne Yates. Boston, MA: Springer US, 1998, pp. 161–182. ISBN: 978-1-4615-5499-8. DOI: 10.1007/978-1-4615-5499-8_10.

[2] BeeGFS. URL: https://www.beegfs.io/ (visited on 10/30/2017).

[3] Janko Böhm, Wolfram Decker, Simon Keicher, and You Ren. *gitfan.lib – A Singular library for computing the GIT fan*. 2016. URL: https://github.com/Singular/Sources (visited on 10/16/2017).

[4] J. Boehm, S. Keicher, and Y. Ren. 'Computing GIT-fans with symmetry and the Mori chamber decomposition of $\overline{M}_{0,6}$'. In: *ArXiv e-prints* (Mar. 2016). arXiv: 1603.09241 [math.AG].

[5] David Gelernter and Nicholas Carriero. 'Coordination Languages and Their Significance'. In: *Commun. ACM* 35.2 (Feb. 1992), pp. 96–107. ISSN: 0001-0782. DOI: 10.1145/129630.376083.

[6] GPI-Space. URL: http://www.gpi-space.de/ (visited on 10/16/2017).

[7] Google Test – *A C++test framework*. URL: https://github.com/google/googletest (visited on 10/19/2017).

[8] Kurt Jensen. 'Coloured Petri nets'. In: *Petri Nets: Central Models and Their Properties*. Ed. by W. Brauer, W. Reisig, and G. Rozenberg. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1987, pp. 248–299. ISBN: 978-3-540-47919-2. DOI: 10.1007/BFb0046842.

[9] Carl Adam Petri. 'Kommunikation mit Automaten'. PhD thesis. Institut für Angewandte Mathematik der Universität Bonn, 1962.

[10] Singular. URL: https://www.singular.uni-kl.de/ (visited on 10/19/2017).

[11] GPI. URL: http://www.gpi-site.com/gpi2/ (visited on 10/16/2017).

List of abbreviations

BLOB	Binary Large Object 14
DRTS	distributed run-time engine 7
Fraunhofer ITWM	*Fraunhofer-Institut für Techno- und Wirtschaftsmathematik* . . 7
PGAS	partitioned global address space 8
STL	Standard Template Library 14
WE	workflow engine 7

Index

Erklärung zu dieser Arbeit

Hiermit erkläre ich, die vorliegende Masterarbeit selbstständig erstellt zu haben. Weiterhin versichere ich, dass sämtliche von mir verwendeten Hilfsmittel und Quellen im Literaturverzeichnis aufgeführt sind.

Christian Reinbold, Hannover, den 11.01.2018