REPORT ON OpenDreamKit DELIVERABLE D5.11

Refactor and Optimise the existing combinatorics SAGE code using the new developed PYTHRAN and CYTHON features.

V. DELECROIX, F. HIVERT

Due on	31/08/2018 (M36)
Delivered on	31/08/2018
Lead	CNRS (CNRS)
Progress on and finalization of this deliverable has been tracked publicly at: `https://github.com/OpenDreamKit/OpenDreamKit/issues/109`	

CONTENTS

1. INTRODUCTION

Computer experimentations in discrete mathematics, in particular enumerative and algebraic combinatorics, require high performance computing. The search spaces are often huge and the best algorithmic strategy for the exploration is not evident and often depend on the answer to the question being probed. Let us cite the following description from [18]:

> Some discrete mathematical problems are embarrassingly parallel, and this has been exploited for years even at Internet scale, e.g. the "Great Internet Mersenne Prime Search". Many parallel algebraic computations exhibit high degrees of irregularity, at multiple levels, with numbers and sizes of tasks varying enormously (up to 5 orders of magnitude). They tend to use complex user-defined data structures, exhibit highly dynamic memory usage and complex control flow, often exploiting recursion. They make little, if any, use of floating-point operations. This combination of characteristics means that symbolic computations are not well suited to conventional HPC paradigms with their emphasis on iteration over floating point arrays.

This deliverable is about experimentations in combinatorics that involve low-level optimization and parallelization as well as the integration of these techniques in the computer algebra system SAGE.

This deliverable has greatly benefited from the OpenDreamKit development workshops *Sage Days 84* held in Olot (Spain) in spring 2017 and *Interfacing (math) software with low level libraries* held in spring 2018 in Cernay-la-Ville (France).

2. TWO EXAMPLES FROM ALGEBRA

To compare computation technologies, our approach was to experiment them on various problems that require intensive computations. We present two such problems.

2.1. **Counting and enumerating integer vectors**

Integer vectors, that is finite sequences of integers, are central in combinatorics as they can be used to encode many different objects. In each situation the integer vectors are subject to various constraints and we will be interested in three different kinds:

- linear constraints (e.g. lower or upper bounds on the entries, maximum differences between adjacent positions, etc.)
- restrictions on the content (e.g. each value should be used at most once)
- symmetries (e.g. lexicographically smallest among all possible cyclic permutations).

One algorithmic problem combinatorics is trying to solve is to provide efficient enumeration of integer vectors subject to these kinds of constraints.

For example, permutations of $\{1, \ldots, n\}$ are encoded by integer vectors on $\{1, \ldots, n\}$ for which each entry appears exactly once. In a sample SAGE session the list can be obtained as follows:

```
sage: P = Permutations(3)
sage: P.list()
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1],
 [3, 1, 2], [3, 2, 1]]
```

Other elementary examples include integer partitions

```
sage: Partitions(5).list()
[[5], [4, 1], [3, 2], [3, 1, 1], [2, 2, 1], [2, 1, 1, 1],
 [1, 1, 1, 1, 1]]
```

or Lyndon words

```
sage: LyndonWords(2,4).list()
[word: 1112, word: 1122, word: 1222]
```

Sometimes, one just wants to count the number of objects in some set and possibly avoid generating the complete list. In SAGE, access to this counting is often implemented by a `cardinality()` method, the implementation of which can often be different from the implementation for enumerating the objects themselves:

```
sage: Permutations(30).cardinality()
265252859812191058636308480000000
sage: Partitions(1000).cardinality()
24061467864032622473692149727991
sage: LyndonWords(2, 120).cardinality()
1107689996487429893125737046788454 6
```

In this deliverable we will demonstrate how the interface to the LaTTe package in SAGE allows more efficient counting and how Cython has helped faster enumeration in SAGE. We will also present some promising experiments involving among other things vectorization techniques (MMX, SSE and AVX instruction sets) and shared memory multicore computing with `Cilk++`.

2.2. **Numerical semigroups**

A computation involving numerical semigroups is known as Frobenius coin problem. Given a set of coins of specifiied denominations, it asks what is the largest monetary sum that cannot be obtained using only coins in that set. Formally,

Definition 1. *A numerical semigroup S is a subset of $\mathbb{N}$ containing 0, closed under addition and of finite complement in $\mathbb{N}$.*

For example the set $S_E = \{0, 3, 6, 7, 9, 10\} \cup \{x \in \mathbb{N}, x \geq 12\}$ is a numerical semigroup. One of the challenging problems in the field of semigroup analysis is to understand how many numerical semigroups there are with some given constraints. To state the question precisely we need a little terminology:

Definition 2. *Let S be a numerical semigroup. We call the genus of S the cardinality of the complementary set $g(S) = \mathrm{card}(\mathbb{N} \setminus S)$.*

For example the genus of the previous example S_E is 6, the cardinality of $\{1, 2, 4, 5, 8, 11\}$.

For a given positive integer g, the number of numerical semigroups of genus g is finite and is denoted by n_g. In J.A. Sloane's on-line encyclopedia of integer sequences [22] we find the values of n_g for $g \leq 52$. These values were obtained by M. Bras-Amorós [4]. It is conjectured that the number of those semigroups grows exponentially with the genus g. So it makes it both interesting and hard to get further values of n_g by directly enumerating semigroups.

Using adequate data structures and parallelization with `Cilk` allowed F. Hivert with his collaborator J. Fromentin to obtain the number of numerical semigroups up to genus $g \leq 70$ and also to confirm for $g \leq 60$ another unrelated famous conjecture due to Wilf [26]. This work has led to the publication [11].

We would like to explain why exploring this tree is quite challenging as a parallel problem. The computation for $g = 70$ involves exploring an extremely large tree with 10^{15} nodes. Of course, when exploring such a tree, different branches can be explored in parallel. However, if the tree is unbalanced, we need a mechanism to rebalance the computation. It appears that the tree of numerical semigroups is extremely unbalanced and is therefore a good prototypical challenge for such a computation. For example, the number of nodes at depth 30 and 45 are $5\,646\,773$ and $8\,888\,486\,816$ respectively. If we sort the nodes at depth 30 decreasingly by their number of descendants at depth 45, then

- The first node has 42% of the descendants;
- The second node has 7.5% of the descendants;
- The 1000 first nodes have 99.4% of the descendants;
- Only $27\,321$ nodes have descendants at depth 45;
- Only 257 nodes have more than 10^6 descendants.

As a consequence, any naive embarassingly parallel algorithm is doomed, as the majority of the time may be spent in just a few branches. There must be some non-trivial load balancing to achieve good performance. We will discuss the different technologies to solve this problem.

2.3. Classes of problems appearing in algebra and combinatorics

In our experiments, we have identified the following four kinds of problems:

(1) Embarrassingly parallel problems: where little or no effort is needed to separate the problem into a number of parallel tasks.
(2) Recursive parallel problems: the computational problem is organized as a recursive tree which can be discovered on the fly during the computation.
(3) Recursive redundant parallel problems: this kind of problem is similar to the previous one except that instead of a tree, the computational problem is organized as a (directed acyclic) graph so that they are multiple different way to reach the same computation. So the computing units have to coordinate to avoid recomputing the same thing too many times or to detect that all the prerequisites for a task are known.
(4) Micro data structure parallel optimization: for many combinatorial structures (permutations, partitions, monomials, Young tableaux), their data can be encoded as a small

sequence of small integers that can often be be handled efficiently by a CPU thanks to vector instructions.

In this work we have mostly focused on point (2) and (4) as (1) is easy and (3) cannot be solved without a good solution for (2) which is still problematic.

3. An overview of techniques and technologies

Before going deeper into the details of this deliverable it is important to draw a general picture of the algorithmic techniques and the hardware and software technologies available.

Let us first distinguish the following two families of programming languages. On the one hand there are the interpreted languages such as Python or Julia. On the other hand we have the compiled languages such as C/C++. The former languages are very convenient for end-users as they provide high level data structures and instructions as well as automatic memory management. This is one of the reasons for the choice for Python as the base language for Sage. However, these higher level languages tend to suffer by comparison to low-level languages in performance, especially when large iterations have to be performed. In this situation, it is necessary to investigate the details of memory allocation and CPU instructions.

A first ingredient of optimization that provides a bridge between interpreted and compiled language is provided by Cython, a Python to C/C++ compiler. Cython plays an important role in this deliverable in two different ways. First, by replacing Python code with Cython code, which generally results immediately in faster code. Second, by developing interfaces to C/C++ libraries.

The use of a low-level language such as C/C++, provides the programmer with a first level of in-core parallelization: the CPU vectorized instructions, i.e. single instructions operating on vectors of multiple data (SIMD). These instructions are also of critical importance for efficient linear algebra algorithms as implemented in the LinBox library. The relevance of these instructions in the context of combinatorics is discussed in Section 4.1.

The second level of parallelization concerns multi-core computations. This is conveniently dealt with by language extensions such as Open MP, Threading Building Block (TBB) or `Cilk`. At this level one important ingredient in optimization is the careful usage of shared memory. Before getting into technical details, we recall in subsection 3.1 the small framework that was implemented in Sage during D5.1: "Turn the Python prototypes for tree exploration into production code, integrate to Sage." that also illustrates the typical problem one faces in combinatorics. More details on the usage of `Cilk` are provided in Section 4.

The last level of parallelism concerns computations with multiple nodes (e.g. multiple computers on a network) and is not addressed by this deliverable.

3.1. Map-reduce and its implementation in Sage

The aim of this section, is to present an example of a typical end-user feature we would like to be able to provide as an actual HPC component. This is a solution for type (2) problems in our classification.

Map-reduce is a general programming model that is shared by parallelized solutions to many problems and is relevant to our situation. In this section we present a small framework implemented in Sage [24] which enables efficient map/reduce-like computations on large recursively defined sets. Map-reduce is a classical programming model for distributed computations where one maps a function on a large data set – applying the same function to each element of the data set – then uses a "reduce" function to summarize all the "map" results. It has a large range of intensive applications in combinatorics:

- Compute the cardinality;
- More generally, compute any kind of generating series;

- Test a conjecture: i.e. find an element of S satisfying a specific property, or check that all of them do;
- Count/list the elements of S having this property.

Use cases in combinatorics often have two specificities. First of all, due to combinatorial explosion, sets often don't fit in the computer's memory or disks and are enumerated on the fly. Then, many problems are flat, leading to embarrassingly parallel computations which are easy to parallelize. However, a second very common use case is to have data sets that are described by a recursion tree which may be heavily unbalanced (as with numerical semigroups described in the previous section).

The framework [10] we developed works on the following input: A **recursively enumerated set** given by:

- the `roots` of the recursion
- the `children` function computing the next level of the recursion
- the `post_process` function, which can also filter intermediate nodes

Then, a **map-reduce problem** is given by:

- the `map` function
- the `reduce_init` function, which specifies an initial value for the reduction process
- the `reduce` function

Here is an example where we count binary sequence of length 15:

```
sage: S = RecursivelyEnumeratedSet( [[]],
....:   lambda l: [l+[0], l+[1]] if len(l) <= 15 else [],
....:   post_process = lambda x : x if len(x) == 15 else None,
....:   structure='forest', enumeration='depth')
sage: sage: S.map_reduce(
....:   map_function = lambda x: 1,
....:   reduce_function = lambda x,y: x+y,
....:   reduce_init = 0 )
32768
```

This framework uses a multi-process implementation of a work-stealing algorithm [3, 3] meaning that if the work is divided unevenly between processes, then when one process becomes idle it can "steal" work from another process's work queue. In the above example of binary sequences it scales as follows

# processors	1	2	4	8
Time (s)	250	161	103	87
Speedup	1	1.55	2.42	2.87

As can be seen on this sample the scaling is far from being linear with the number of cores. This is explained by the increase of job stealing coming with parallelization. A challenge will be to reduce this overhead and achieve linear scaling.

Though it doesn't really qualify as HPC, principally due to the use of Python. Tough it allowed us to efficiently parallelize dozens of experiments ranging from Coxeter group and representation theory of monoids, to the combinatorial study of the C3 linearization algorithm used to compute the method resolution order (MRO) in scripting languages such as PYTHON and Perl [25].

4. LOW-LEVEL EXPERIMENTS

We now present the result of the evaluation of various parallelization technologies applied to algebraic combinatorics computations.

4.1. **Combinatorial structures and vector instructions**

Current SIMD instruction sets were introduced in three stages: MMX (set of single instruction multiple data instruction set introduced in 1997), SSE (Streaming SIMD Extensions, introduced in 1999) and more recently AVX (Advanced Vector Extensions, introduced in 2008).

In many combinatorial structures (permutations, partitions, monomials, Young tableaux), the data can be encoded as a small sequence of small integers that can often be handled efficiently thanks to these vector instructions. For example, on current desktop CPU architectures (x86), small permutations ($N \leq 16$) are very well handled. Indeed, thanks to machine instructions such as `PSHUFB` (Packed Shuffle Bytes), applying a permutation on a vector only takes a few CPU cycles. Here are some examples of operations with their typical speedups via SIMD instructions:

Operation	Speedup
Sum of a vector of bytes	3.81
Sorting a vector of bytes	21.3
Inverting a permutation	1.97
Number of cycles of a permutation	41.5
Number of inversions of a permutation	9.39
Cycle type of a permutation	8.94

Unfortunately, from a developer's point of view, this requires rethinking all the algorithms, and there is nearly no support from the compiler for automating this task.

As a part of the OpenDreamKit deliverable, we started to develop a new library called `HPCombi`[1]. The goal is to use SSE and AVX instruction sets for very fast manipulation of combinatorial objects of small sizes. It is still in an experimental stage and currently deals only with permutations, transformations and partial transformations. We also have experimental code for partitions and boolean matrices.

Despite its infancy the code is already used by libsemigroups [19] (which deals with a different kind of semigroup from numerical semigroups) by James Mitchell. It is a C++ library for semigroups and monoids using C++11, and is used in the Semigroups package for GAP. The development version is available on GitHub, and there are Python bindings which makes it usable from Sage. The development of a more thorough Sage interface is planned.

Finally, we want to stress out that this is actually a research problem. Indeed, except for sorting a vector where we used a classical sorting network algorithm, all the operations in the previous table of operations require the design of a new algorithms as nearly no combinatorial algorithms where conceived with vector instructions in mind. For example, for the simple task of inverting a permutation, we designed 4 different new algorithms:

- `inverse_sort`: a classical sort algorithm
- `inverse_search`: using parallel binary search. Limited by the current size of vector registers (16 bytes) which support arbitrary array manipulation (in particular the shuffle instruction), it is not the fastest algorithm. Yet it is the only one of logarithmic complexity, so it should become the fastest if and when a CPU supporting larger registers for this instruction become available;
- `inverse_power`: a binary powering algorithm;
- `inverse_cycle`: using the cycle decomposition. It is currently the fastest.

Preliminary results from this work were presented as a keynote invited talk at the 8th International Workshop on Parallel Symbolic Computation (PASCO 2017), Kaiserslautern, Germany, July 23-24, 2017 and at the Scottish Programming Languages Seminar, 5th June 2018, Heriot-Watt University. A research paper on this subject is in preparation as a result of this work.

[1] `https://github.com/hivert/HPCombi`

4.2. **Combinatorial structures and GPU computations**

In the previous section we presented how the vector instruction of modern CPUs enabled a large speedup for small combinatorial objects. The main requirement is that the datastructure fits in one or a handful of the CPU registers (data storage areas internal to the CPU, where intermediate results of computations are stored). However, while this covers a lot of practical case on algebraic combinatorics, there is sometimes a need for larger combinatorial objects, due to combinatorial explosion. One lead was to investigate if graphics processing units (GPUs) could speed up these kinds of computations. Our benchmark was to write a toy implementation of the algorithm used in libsemigroups [19] replacing the use of HPCombi and AVX CPU vector instruction by some GPU code. These algorithms enumerate the elements of a transformation of semigroups.

This kind of computation typically involves the repetition of two computation stages:

(1) compute the composition of large functions on a discrete set;
(2) store them in a hash table to remove duplicates.

It should be noted that the first stage doesn't require any arithmetic, but amount to shuffling large chunks of data in memory. This is a very atypical usage of a GPU which is tailored to do arithmetic on larger vectors of floating point numbers, and it was very difficult to predict the behavior. One expected cause of this inconsistency is the communication time between the CPU and the GPU.

The outcome of the experiments lead by Daniel Vanzo (research engineer at LRI/UPSud) under the supervision of Florent Hivert were that:

- Composition of large functions can be greatly accelerated (typically $\times 100$) by the GPU, provided that the CPU ask the GPU to perform enough of them (more than 1K) at once.
- The same is true for the computation of hash values.

Although this requirement seems reasonable for the first stage, it is not for the second: in a reasonable use case, one needs to store the results of the computation in a large hash table which typically is at the limit if not larger than the size of the GPU's memory. We therefore decided to keep the hash table in the computer's main memory, external to the GPU, degrading performance due random memory accesses. Nearly all memory accesses triggered a cache miss which resulted in slowdown of as much as $60\times$.

The conclusion is that in order to achieve a useful speedup with the GPU, the entire computation must be hosted inside the GPU's internal memory. This has two very important drawbacks:

- It drastically limits the size of the computation as CPU memory tends to be around $10\times$ larger than the GPU's.
- Since GPU programming is not a widespread skill, it forces us to write seemingly black-box, monolithic algorithms.

This second point doesn't fit well within the research-driven use cases in the framework of the OpenDreamKit project; that is, it is difficult to understand or modify by a typical researcher. We think that this technology is not directly applicable in algebraic combinatorics in general. Though, it remains interesting for very specific computations where we are ready to pay the price of writing specific and poorly reusable programs.

4.3. **Numerical semigroups**

The computation of numerical semigroup is a good challenge for large tree explorations (case (2) in our classification). Different branches of the tree can be explored in parallel by different CPU cores. The delicate part is to ensure that all cores are working, i.e. always have a new branch to explore.

What makes this problem particularly difficult is the size of the trees (typically up to 10^{15} in our experiments) and the granularity of the computation (typically only $10 - 50$ns is spent on each tree node). This is extremely demanding on the load balancing algorithms and their implementations.

The clear algorithmic solution is to use work stealing algorithms as described in [2, 3]. Let us recall that a prototypical Python implementation with a map-reduce fronted is in Deliverable D5.1 [10]. However, it was clear from the beginning that this implementation was a tool to help rapid prototyping in day-to-day research, but it wasn't meant for a high performance computing use case. Hence, we had to experiment with a few lower-level technologies. We present them here together with their behavior in our prototypical examples:

- The `Cilk++` [23] technology is particularly well suited for those kinds of problems. For our computation, we used the free version which is integrated into the GNU C compiler [15] since version 5.8.

  `Cilk` is a general-purpose language designed for multithreaded parallel computing. The `C++` incarnation is called `Cilk++`. The main principle behind the design of the `Cilk` language is that the programmer should be responsible for exposing the parallelism, identifying elements that can safely be executed in parallel; the run-time environment decides during execution how to actually divide the work between CPU cores. The parallel features of `Cilk++` are used mainly through the `cilk_spawn` keyword: used on a procedure call, it indicates that the call can safely operate in parallel with the remaining code of the current function. Note that the scheduler is not obliged to run this procedure in parallel; the keyword merely alerts the scheduler that it can do so.

  Overall, this makes `Cilk++` very easy to use, with short and very readable source code. Moreover, it turns out that `Cilk++` is extremely good at solving the problems we were interested in. To provide some figures of the performance we managed to achieve in counting monoids–a problem similar to that of counting semigroups. We performed a full exploration of the tree up to a depth of 70 on a 32 core Haswell CPU at 2.3 Ghz. The number of monoids at depth 70 is 1607394814170158. It tooks $2.528 \cdot 10^6$ s (29 days and 6 hours) exploring $2590899247785594 = 2.59 \cdot 10^{15}$ monoids at a rate of $1.02 \cdot 10^9$ monoids per second. Each monoid is stored in 240 bytes. Storing all the computed monoids would take $6.22 \cdot 10^{17}$ bytes of data, which means that we generated $2.46 \cdot 10^{11}$ bytes of data per second.

  The main drawback, is that both Intel and the GCC team decided to **deprecate and no longer maintain the `Cilk++` extension**. So we looked for alternatives.
- OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior. It allows to spawn tasks using special pragma directives on the compiler.

  Unfortunately, for computations like numerical semigroups, there is a huge number of small tasks that are spawned. We used the implementation from the GCC compiler. At the time of our experiment, we found that the scheduler doesn't scale to this huge number of tasks required. The scheduling overhead was several order of magnitude larger than `Cilk++` making it unusable for these kinds of computations.
- Intel's Threading Building Blocks is a C++ template library developed by Intel for parallel programming on multi-core processors. Using TBB, a computation is broken down into tasks that can run in parallel. The library manages and schedules threads to execute these tasks.

  However, it suffers exactly the same problem as OpenMP: the scheduler doesn't scale for huge numbers of tasks. Intel's FAQ reads, "A good rule of thumb is that an Intel TBB

application should have approximately 10 tasks for every worker." Similarly, in [17], they consider a problem requiring micro-task computations of roughly 30k tasks, each of which takes 4k CPU cycles. Whereas in the numerical monoid problem task takes around 60 CPU cycles, the number of tasks easily exceeds 1 billion for a 1 minute long computation.

- Developping efficient scheduler able to deal with very large amount of lightweight tasks is an active research area. For instance the ongoing PhD thesis by Blair Archibald under the supervision of Phil Trinder at University of Glasgow exactly also aims to distribute the computation on a cluster of machines. As a result, they developed a C++ library called YewPar [1] which they present as "A Collection of High Performance Parallel Skeletons for Tree Search Problems" using the well established HPX [14] parallel library/runtime.

  By the time we got in contact with them, they had already decided to use our numerical semigroup algorithm as a benchmark for their scheduling implementation. We think that this is a good indication that our choice was correct. Though not as easy as `Cilk++`, the YewPar's integration provides distribution across several machines. If only used on one multicore machine, YewPar is more or less only 2.5 times slower than `Cilk++`. The main question is long term maintenance: the website says "This library is currently experimental and should be considered very unstable," and being the work of a PhD student, there is no warranty that it will still be maintained in a few years.
- We also advised three master student interns; namely, Adrien Pavão, Thomas Foltête, and Edgar Fournival to explore the Spark technology and the Go language. It turned out that these technology weren't fit for our needs. On the other hand, their work led to discovery of a basic bug in the GCC implementation of `Cilk++` [13].

In conclusion, for large combinatorial tree explorations our current recommendation is to use `Cilk++` which is both very efficient and simple to use and learn. We hope that an eventual alternative will manage to achieve similar performance. A promising newcomer is the Cilk Hub initiative[2] to continue maintaining Cilk under a new project, but we have not yet had ocassion to seriously experiment with it.

5. Improvements to the Sage platform

5.1. **Polytopes and linear programming**

As mentioned in the introduction, see 2.1, integer vectors are omnipresent in combinatorics. In many situations, the combinatorial constraints are linear equalities or inequalities. For example, the partitions of n are given by non-negative integer vectors $(x_1, \dots, x_n)$ in $\mathbb{R}^n$ so that $x_1 \geq x_2 \dots x_n$. In other words, some general linear programming techniques come into play. Improving Sage's capabilities in polytope computations and linear algebra is critical for combinatorics.

Many libraries, often written in C/C++ exist and one of our tasks was to create or improve the existing Sage interfaces to those libraries. Most of these interface rely on Cython to provide a bridge between Python and C/C++. Let us mention the creation of the stand-alone library pplpy [9] which now provides access to the PPL library to any Python user, and is integrated into Sage.

At a higher level of interaction, an interface to LaTTe in Sage has been developed (see trac tickets [6] and [8]). It allows efficent counting of integral points in polytopes. The interface is completely transparent to the user as can be seen in the following Sage session:

```
sage: n = 10
sage: I1 = [[0] + [0]*i + [1] + [0]*(n-1-i)
....:       for i in range(n)]
```

[2] `http://cilk.mit.edu/`

```
sage: I2 = [[0] + [0]*i + [1,-1] + [0]*(n-2-i)
....:       for i in range(n-1)]
sage: P = Polyhedron(ieqs=I1 + I2, eqns=[[-n] + [1]*n])
sage: P.integral_points_count()
42
```

Let us also mention the SAGE interface to the Polymake software [12] which allows direct interaction from SAGE. Polymake is a reference software for Polyhedral computations and its inclusion in SAGE has been greatly beneficial.

Polyhedral computations are not restricted to rational numbers. For a long time floating point polyhedral libraries have existed. However, they are often not satisfactory, as floating-point rounding errors can lead to subtle contradictions. In the framework of OpenDreamKit, V. Delecroix started a C/C++ library for computations with embedded number fields called e-antic [7]. It allows exact computations over algebraic numbers. This has successfully been included in the Normaliz [5] software and it is now possible to use it to very efficiently construct polytopes over number fields. We aim to finalize the inclusion of this code and provide a SAGE interface to it.

5.2. **Cythonization of combinatorics code**

A huge amount of code in SAGE is written in plain Python; this is in particular true for a large amount of the combinatorics code. Thanks to Cython, one can easily gain speed-up in computations. A lot of effort has been made to improve the performance of the current code. We emphasize some of the efforts that has been made in OpenDreamKit.

Two examples of successful "cythonization" were achieved for permutations [21] and Lyndon words [20]. These combinatorial objects were already mentioned in the introduction; see 2.1. The challenge was to write iterators with maximal efficiency together with a simple Python wrapper intended for SAGE users. While in both situations presented here, efficient algorithms exist we have focused on three levels of optimization:

- *basic cythonization*. Cython does a decent job in direct optimization with little human intervention.
- *in-place iteration*. As already mentioned, combinatorial objects are typically represented by integer vectors. Dealing with a lot of plain lists in Python has a huge cost in memory allocation. The most natural workaround is to provide an in-place iterator, that is, an iterator that modifies the data without creating new objects.
- *use C arrays*. Finally, using a Python data structure closer to C arrays than Python's built-in "list" type allows one to gain another speedup factor.

Below we present simple timings of the iteration through all permutations of $\{1, 2, \ldots, n\}$ with three runs of the same algorithm. It consists in modifying a given permutation into the next one for the lexicographic order. The three columns represent the different versions: in Python using Python lists, in Cython using Python lists and finally in Cython using C arrays.

n	Python	Cython on lists	Cython on arrays
9	410ms	55ms	37ms
10	2.9s	309ms	163ms
11	32s	3.2s	1.68s
12	$>$ 2 min	36.3s	19.5s

These experimets show a near 10x speedup when passing to Cython (with almost no modification), then Using arrays instead of lists provides an additional 2x speedup.

Let us also mention a slightly different work related to combinatorics: Dancing links is the name of an algorithm that finds all solutions to the so-called exact cover problem. It can efficiently be used to solve tiling problems. Using an embarrassingly parallel method, this naive

parallelization approach allowed us to obtain the list of solutions 2 to 3 times faster using a 4 core machine (see [16]).

6. Conclusion

SAGE is a general purpose computer algebra system with a lot of code related to combinatorics. The Python language has some weaknesses regarding speed and the aim of the deliverable was to circumvent these weaknesses. As we demonstrated, using Cython and careful memory usage allows us to provide an important speed up in enumeration. Cython is at the same time an ideal tool to provide access to efficient implementations in C/C++ from Python.

Very promising experimentations have been performed in C regarding the usage of vectorized operations (SIMD) as well as efficient parallelization. We will pursue the development of the HPCombi library with the aim to get it integrated into different computer algebra system such as SAGE and GAP.

References

[1] Blair Archibald. A Collection of High Performance Parallel Skeletons for Tree Search Problems. 2018. URL: https://github.com/BlairArchibald/YewPar.

[2] R. D. Blumofe and C. E. Leiserson. "Scheduling Multithreaded Computations by Work Stealing". In: Foundations of Computer Science, Annual IEEE Symposium on 0 (1994), pp. 356 –368. DOI: http://doi.ieeecomputersociety.org/10.1109/SFCS.1994.365680.

[3] R. D. Blumofe and C. E. Leiserson. "Scheduling Multithreaded Computations by Work Stealing". In: J. ACM 46.5 (1999), pp. 720–748.

[4] M. Bras-Amorós. "Fibonacci-like behavior of the number of numerical semigroups of a given genus". In: Semigroup Forum 76.2 (2008), pp. 379–384. ISSN: 0037-1912. DOI: 10.1007/s00233-007-9014-8.

[5] W. Bruns et al. Normaliz. Algorithms for rational cones and affine monoids. https://www.normaliz.uni-osnabrueck.de/.

[6] V. Delecroix. Computing Ehrhart polynomials with LattE. https://trac.sagemath.org/ticket/18211.

[7] V. Delecroix. e-antic. Exact computations with embedded number fields. https://github.com/videlec/e-antic.

[8] V. Delecroix. generic latte_int interface to count. https://trac.sagemath.org/ticket/22497.

[9] V. Delecroix. pplpy. An Python interface to the Parma Polyhedra Library. https://gitlab.com/videlec/pplpy.

[10] J.-B. Priez F. Hivert and N. Cohen. Parallel computations using RecursivelyEnumeratedSet and Map-Red The Sage Development Team. 2016. URL: http://doc.sagemath.org/html/en/reference/parallel/sage/parallel/map_reduce.html.

[11] Jean Fromentin and Florent Hivert. "Exploring the tree of numerical semigroups". In: Math. Comput. 85.301 (2016), pp. 2553–2568. DOI: 10.1090/mcom/3075. URL: https://doi.org/10.1090/mcom/3075.

[12] E. Gawrilow and Joswig M. Polymake: a framework for analyzing convex polytopes. https://polymake.org/. 1997.

[13] GNU. Random segfault using local vectors in Cilk function. GCC Bugzilla. 2016. URL: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=80038.

[14] The STELLAR Group. The C++ Standard Library for Parallelism and Concurrency. 2018. URL: http://stellar-group.org/libraries/hpx/.

[15] B.V. Iyer, R. Geva, and P. Halpern. "Cilk™ Plus in GCC". In: GNU Tools Cauldron. 2012. URL: http://gcc.gnu.org/wiki/cauldron2012?action=AttachFile&do=get&target=Cilkplus_GCC.pdf.

[16] S. Labbé. dancing links: find all solutions in parallel. https://trac.sagemath.org/ticket/25125.

[17] S. Lu and Q. Li. "Improving the Task Stealing in Intel Threading Building Blocks". In: 2011 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery. 2011, pp. 343–346. DOI: 10.1109/CyberC.2011.61.

[18] Patrick Maier et al. "High-Performance Computer Algebra: A Hecke Algebra Case Study". In: Euro-Par 2014 Parallel Processing. Ed. by Fernando Silva, Inês Dutra, and Vítor Santos Costa. Cham: Springer International Publishing, 2014, pp. 415–426. ISBN: 978-3-319-09873-9.

[19] James Mitchell and M. Torpey. C++ library for semigroups and monoids. 2018. URL: https://github.com/james-d-mitchell/libsemigroups.

[20] T. Scrimshaw. Implement faster iterator for Lyndon words. https://trac.sagemath.org/ticket/26111.

[21] T. Scrimshaw. Implement iterator for generic permutations in Cython. https://trac.sagemath.org/ticket/23734.

[22] N. J. A. Sloane. The On-Line Encyclopedia of Integer Sequences. http://oeis.org/. 2014.

[23] Software.intel.com. Intel® Cilk™ Homepage. https://www.cilkplus.org/. 2013.

[24] W. A. Stein et al. Sage Mathematics Software (Version 7.6). The Sage Development Team. 2016. URL: http://www.sagemath.org.

[25] Nicolas M. Thiéry. The C3 algorithm, under control of a total order. The Sage Development Team. 2016. URL: http://doc.sagemath.org/html/en/reference/misc/sage/misc/c3_controlled.html.

[26] H. S. Wilf. "A circle-of-lights algorithm for the "money-changing problem"". In: Amer. Math. Monthly 85.7 (1978), pp. 562–565. ISSN: 0002-9890.