# REPORT ON OpenDreamKit DELIVERABLE D3.11

## HPC enabled SAGE distribution

ALEXIS BREUST, KARIM BELABAS, JEAN-GUILLAUME DUMAS, JEROEN DEMEYER, WILLIAM B. HART, STEVE LINTON, CLÉMENT PERNET, REIMER BEHRENDS, HONGGUANG ZHU


| | |
|---|---|
| Due on | 31/08/2019 (M48) |
| Delivered on | 03/09/2019 |
| Lead | Université Grenoble Alpes (UGA) |
| Progress on and finalization of this deliverable has been tracked publicly at: https://github.com/OpenDreamKit/OpenDreamKit/issues/60 | |

DELIVERABLE DESCRIPTION, AS TAKEN FROM GITHUB ISSUE #60 ON 2019-09-03

- **WP3:** Component Architecture
- **Lead Institution:** Univ. Grenoble Alpes
- **Due:** 2019-08-31 (month 48)
- **Nature:** Other
- **Task:** T3.5 (#54)
- **Proposal:** p. 42
- **Final report** (sources)

The primary use of computational mathematics software is to perform experimental mathematics, for example for testing a conjecture on as many as possible instances of size as large as possible. In this perspective, users seek for computational efficiency and the ability to harness the power of a variety of modern architectures. This is particularly relevant in the context of the OpenDreamKit Virtual Research Environment toolkit which is meant to reduce entry barriers by providing a uniform user experience from multicore personal computers -- a most common use case -- to high-end servers or even clusters. Hence, in the realm of this project, we use the term High Performance Computing (HPC) in a broad sense, covering all the above architectures with appropriate parallel paradigms (SIMD, multiprocessing, distributed computing, etc).

Work Package 5 has resulted in either enabling or drastically enhancing the high performance capabilities of several computational mathematics systems, namely `Singular` (D5.13 #111), `GAP` (D5.15 #113) and `PARI` (D5.16, #114), or of the dedicated library `LinBox` (D5.12 #110, D5.14 #112).

Bringing further HPC to a general purpose computational mathematics system such as `SageMath` is particularly challenging; indeed, they need to cover a broad -- if not exhaustive -- range of features, in a high level and user friendly environment, with competitive performance. To achieve this, they are composed from the ground up as integrated systems that take advantage of existing highly tuned dedicated libraries or sub-systems such as aforementioned.

Were report here on the exploratory work carried out in Task to expose HPC capabilities of components to the end-user level of an integrated system such as `SageMath`.

Our first test bed is the `LinBox` library. Its multicore parallelism features have been successfully integrated in `Sage`, with a simple API letting the user control the desired level

of parallelism. We demonstrate the efficiency of the composition with experiments. Going beyond expectations, the outcome has been integrated in the next production release of `SageMath`, hence immediately benefiting thousands of users.

We proceed by detailing the unique challenges posed by each of the `Singular`, `PARI`, and `GAP` systems. The common cause is that they were created decades ago as standalone systems, represent hundreds of man-years of development, and were only recently re-designed to be usable as a parallel libraries. Some successes were nevertheless obtained in experimental setups and pathways to production are discussed.

We conclude with lessons learned at the occasion of this work and through expertise sharing within and beyond OpenDreamKit: levels of integration one may wish for when composing parallel computational mathematics software. and challenges such integration would raise.

CONTENTS



## 1. EXPOSING PARALLEL FEATURES OF COMPONENTS IN SAGEMATH

### 1.1. LINBOX's parallel finite field linear algebra

We have been successful in exposing the parallel features of the LINBOX library, and more specifically of its kernel for finite field linear algebra, `fflas-ffpack`. By nature, a library is designed with composability in mind, and thread safety is among the required features a parallel library should provide.

1.1.1. *Context.* Finite field linear algebra is a core building block in computational mathematics. It has a wide range of applications, including number theory, group theory, combinatorics, etc. SAGEMATH relies on the FFLAS-FFPACK library for its critical linear algebra operations on prime fields of less than 23 bits and consequently also for numerous computations with multi-precision integer matrices.

The FFLAS-FFPACK library had some preliminary support for multi-core parallelism for matrix multiplication and Gaussian elimination. Instead of being tied to a specific parallel language, the library uses a Domain Specific Language, Paladin [3], to provide the library programmer with a unique API for writing parallel code, which is then translated into OpenMP [5], Cilk [1], Intel-TBB [4], or XKaapi [2] directives. Beside portability and independence from a given technology, this also makes it possible to benchmark and compare how parallel runtimes perform. This is particularly important here, since many of the compute intensive routines of FFLAS-FFPACK share specificities that are often challenges for parallel runtime:

**Recursion:** by design, sub-cubic linear algebra algorithms are recursive and so are most routines in the library.

**Heterogeneity:** many exact computations must deal with data with size unknown before their actual computation. For instance rank deficiencies in Gaussian elimination may

generates a block decomposition of varying dimensions and therefore computing tasks will have heterogeneous load.

**Fine grained task parallelism:** the combination of the two above constraints leads to the consideration of recursive task fine-grained parallelism, such that a work-stealing engine could efficiently balance the heterogeneity. However, recursive tasks have been for a long time rather inefficient in e.g. OpenMP implementations, and the ability to handle numerous small tasks is also demanding on parallel runtimes.

1.1.2. *Integration within* SAGEMATH. The main tasks for the exposition of the parallel routines of FFLAS-FFPACK in SAGEMATH were the following:

(1) improving existing parallel code for Gaussian elimination and matrix multiplication in the FFLAS-FFPACK library;
(2) adding new parallel routines in FFLAS-FFPACK for the most commonly used operations in SAGEMATH: the determinant, the echelon form, the rank, and the solution of a linear system;
(3) connecting these parallel routines in SAGEMATH providing the user with a precise control on the number of threads allocated to the linear algebra routines.

The first two items involved 15 pull-requests, merged and released in `fflas-ffpack-2.4.3`[1]. This release was produced simultaneously with that of the two other libraries in the LINBOX ecosystem: `givaro-4.1.1`[2] and `linbox-1.6.3`[3] then integrated into SAGEMATH in tickets

- https://trac.sagemath.org/ticket/26932 and
- https://trac.sagemath.org/ticket/27444,

which will appear in release 8.9 of SAGEMATH.

As a side note, the integration of these new releases including the contributions to D5.12 led to a significant speed-up in the sequential computation time of finite field linear algebra in SAGEMATH, as show in Table 1.

| | $\mathbb{Z}/4\,194\,301\mathbb{Z}$ | | $\mathbb{Z}/251\mathbb{Z}$ | |
|---|---|---|---|---|
| | Before | After | Before | After |
| Matrix product | 3.61 | 3.57 | 1.59 | 1.5 |
| Determinant | 2.96 | 1.52 | 1.54 | 0.731 |
| Echelon form | 3.59 | 1.86 | 1.82 | 0.692 |
| Linear system | 8.9 | 5.13 | 3.7 | 1.79 |

TABLE 1. Improvement of the sequential code with `fflas-ffpack-2.4.3`. Computation time in seconds for a $4000 \times 4000$ machine over a 22 bits and a 8 bits finite field, on an Intel i7-8950 CPU.

For Item (3), we explored several options and chose to rely on and extend the Singleton class `Parallelism` in SAGEMATH. This class works as a dictionary registering the number of threads with which each component in SAGEMATH can run in parallel.

For example, the following code requires than any linear algebra routine relying on linbox be parallelized on 16 cores.

[1] https://github.com/linbox-team/fflas-ffpack/releases/tag/2.4.3
[2] https://github.com/linbox-team/givaro/releases/tag/4.1.1
[3] https://github.com/linbox-team/linbox/releases/tag/v1.6.3

```
sage: Parallelism().set("linbox",16)
```

The following session demonstrates the gain in parallelizing the product of a random $8000 \times 8000$ matrix over $\mathbb{Z}/65521\mathbb{Z}$ with itself using 16 cores:

```
pernet@dahu34:~/soft/sage$ ./sage
SageMath version 8.9.beta8, Release Date: 2019-08-25
sage: a=random_matrix(GF(65521),8000)
sage: Parallelism()
Number of processes for parallelization:
 - linbox computations: 1
 - tensor computations: 1
sage: time b=a*a
CPU times: user 17.5 s, sys: 1.04 s, total: 18.5 s
Wall time: 18.5 s
sage: Parallelism().set("linbox",16)
sage: Parallelism()
Number of processes for parallelization:
 - linbox computations: 16
 - tensor computations: 1
sage: time b=a*a
CPU times: user 28.9 s, sys: 4.85 s, total: 33.8 s
Wall time: 2.41 s
```

Figure 1 shows computation time of three high level sage routines `b=a*a`, `a.determinant()` and `a.echelon_form()` on a large square matrix of order 20000, with varying number of cores. The speed-up relative to a single threaded run of these timings is reported in Figure 2

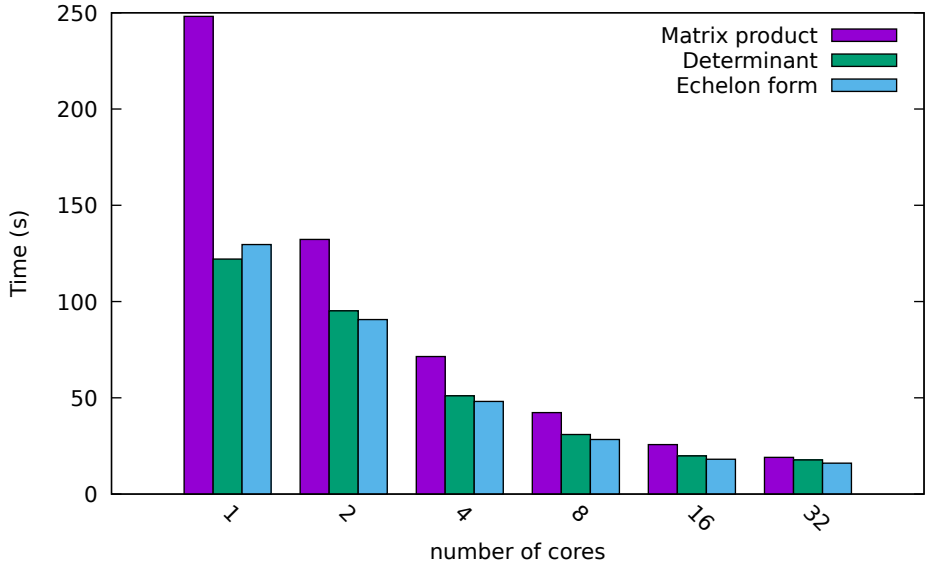


FIGURE 1. Parallel computation time for some finite field linear algebra operations in SageMath on a 32 core Intel Xeon 6130 Gold. Matrices are $20\,000 \times 20\,000$ with full rank over $\mathbb{Z}/1\,048\,573\mathbb{Z}$.

The scalability shown on Figure 2 is good but not close to the best theoretical linear speedup. This happens for the following reasons:

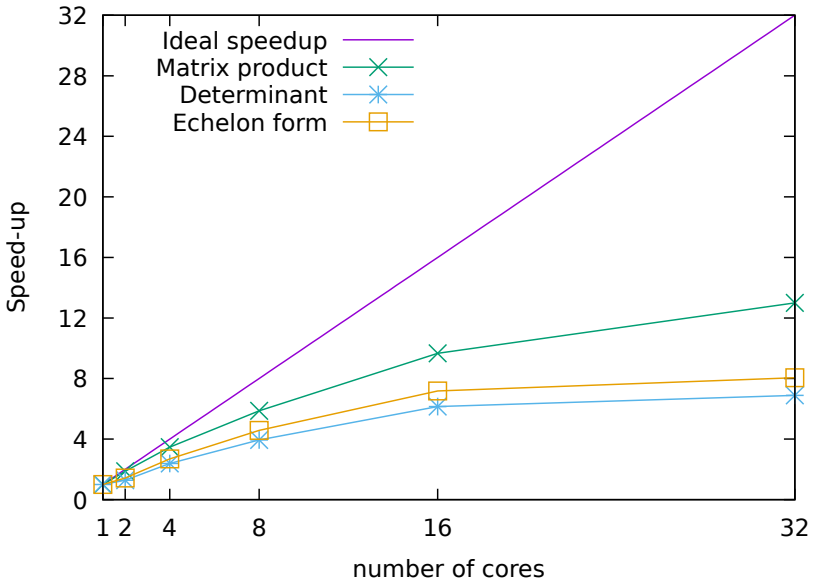


FIGURE 2. Parallel speedup for some finite field linear algebra operations in SageMath on a 32 core Intel Xeon 6130 Gold. Matrices are $20\,000 \times 20\,000$ with full rank over $\mathbb{Z}/1\,048\,573\mathbb{Z}$.

- first, the interface between the system SAGEMATH (running an interactive ipython) and the compiled code of the library adds a constant overhead despite our efforts to reduce it as much as possible. According to Amdhal's law, such an overhead severely penalizes the speedup measured;
- the use of sub-cubic arithmetic for the sequential matrix product tasks implies that the workload increases with the number of cores. Therefore, the true ideal speed-up curve should lie slightly below the main diagonal;
- we did not disable the turbo-boost on this server. Consequently, runs on few cores are likely executed at a higher clock frequency, hence penalizing the speedup for large numbers of cores.

Still, these timings show that very high performances can be attained directly from the high level interface of SAGEMATH, for instance the computation speed on 32 cores reaches 838Gfops[4] for matrix multiplication and 301 Gfops for the determinant.

### 1.2. Multi-threaded SINGULAR

In D5.13 we report on parallelisation of multivariate polynomial arithmetic in SINGULAR. This is achieved by outsourcing multivariate polynomial arithmetic to the C FLINT library and implementing parallelism there using a data representation optimised for that purpose (SINGULAR's primary data representation of multivariate polynomials is in contrast optimised for Gröbner basis computations, which can themselves be components of a parallel computation using SINGULAR's parallel library infrastructure).

As of now, the new parallel multivariate polynomial implementation is available in the latest development version of FLINT and SINGULAR. This will have clear benefits for the SINGULAR

[4] 1Gfops $= 10^9$ field operations per second.

community and VREs built around SINGULAR itself, e.g. via the Jupyter front end for Singular developed as part of D4.4: "Basic JUPYTER interface for GAP, PARI/GP, SAGE, Singular".

The strategy for exposure in SAGEMATH is that this will be automatic the next time the version of SINGULAR is updated in SAGEMATH. This is possible as the improvements are accessed through already existing interfaces.

An alternative for the SAGEMATH project is to wrap the new multivariate interface of FLINT directly. However this would be a significant undertaking dependent on the SAGEMATH community itself. FLINT is already a component of SAGE, but at the library level, the new implementation is a completely new, and quite extensive, interface.

It is expected that the next production version of SINGULAR will be due in the coming winter under SINGULAR's approximate yearly schedule.

The step of updating the version of the SINGULAR computer algebra system supported in SAGEMATH is normally taken care of by the SAGEMATH community. The effort required is proportional to the number of changes in the SINGULAR interface. As a computer algebra system, SINGULAR is much more extensive than a narrowly focused library, therefore this effort can still be non-trivial. It should be noted that this effort is not impacted by the new parallel multivariate implementation as we have been successful in not changing the SINGULAR interface to support it. However, all the other usual work of updating the SINGULAR version, with its extensive interface that touch many aspects of SAGEMATH still needs to be carried out at the next release.

Thanks to independent work, the next production release of SINGULAR will also include one or more of the following: fast rational functions, factorisation, additional polynomial orderings and Gröbner basis speedups that directly build on the new parallel multivariate arithmetic. The work to speed up multivariate arithmetic in SINGULAR is complete as part of D5.13 and independent work to speed up rational functions is almost complete. It can thus be expected that SAGEMATH users will benefit from all these new features at the time of the next SINGULAR update in SAGEMATH.

SINGULAR also independently provides other mechanisms for parallel code, e.g. via its parallel library and through the HPC Singular project, which provides parallelism at a coarser grain than the fine grained parallelism implemented in the new multivariate arithmetic. SAGEMATH already has a well-developed interface for accessing SINGULAR libraries and SAGEMATH developers can already benefit from this, due to the fact that anything that works at the SINGULAR interpreter prompt also works in SAGEMATH.

We refer to D5.13 for tables showing the speedups expected at the SINGULAR level and a discussion of the differences with a direct interface to FLINT.

### 1.3. **Multi-threaded PARI**

As reported in D5.16, the number theory library PARI supports multi-threading for various operations. SAGE uses PARI for much of its number theory functionality. As SAGE already interfaces PARI, a similar situation exists for PARI-MT (the multi-threading-enabled configuration of PARI) in SAGEMATH as for SINGULAR. As soon as the next update of the version of PARI in SAGEMATH is completed, parallelism can be enabled in PARI-MT.

Some experiments have already been conducted in this direction and the PARI-MT interface can be made to work in SAGEMATH with minimal effort under the standard assumption that it is not itself called from multiple threads not under the control of PARI.

For example, the documentation builder of SAGE currently uses PARI to create images for documentation. However, to handle the hundreds of pages of the documentation – it uses Python's multiprocessing module, which (perhaps confusingly) also uses multiple threads.

We expect that this problem can be easily solved with some additional effort in either the SAGE–PARI interface or within PARI itself, for example by enabling run-time configuration of whether to use sequential or parallel computation in PARI.

Nevertheless, PARI-MT is usable in SAGE if one is careful not to build the documentation in this mode, nor to call PARI functions from SAGE-level threads. It is expected that remaining issues will be resolved in a future SAGEMATH release.

We refer again to D5.16 for details on the expected benefits, which should translate directly to SAGEMATH users, due to the straightforward strategy employed: speed up PARI through threading without changing the interface.

### 1.4. HPC GAP

HPC GAP has been in development for very many years, and is now esssentially stabilised in the main GAP repository through an extensive recent effort of the GAP community. It is even able to be used with a small patch applied to an *existing* GAP release.

It is not currently possible to use HPC GAP through the `libgap` library, however independent work is currently being undertaken by the main developer of HPC GAP to make this possible. This is the preferred way of accessing the GAP system from SAGEMATH.

For the purposes of an HPC enabled SAGE distribution, HPC GAP should perhaps better be viewed as a standalone application, distinct from GAP itself. This may even remain the case into the foreseeable future, as single threaded performance is lower in HPC GAP than GAP itself, and compile time flags are required to enable HPC GAP, as a result. Further tuning may eventually alleviate some of this.

There are already some GAP projects that have taken advantage of HPC GAP, though the tool is very new for the GAP community. It is expected that the GAP community will develop many novel uses that benefit from parallelisation. These could be interfaced by the SAGEMATH community in the future, at least for very large, specialised projects and parallel computations that cannot be carried out with the existing serial project.

For further details on `libgap` and HPC GAP, see D5.15.

## 2. CHALLENGES OF COMPOSING PARALLEL SOFTWARE SYSTEMS

As mentioned above, we have been successful in either integrating improved libraries or interfacing to the external computer algebra systems in most cases without adding to the maintenance burden of SAGEMATH.

We now summarise the current situation before discussing what we have learned through the exchange of expertise about further enhancing SAGEMATH's HPC capabilities:

- It is to be expected that SINGULAR's parallel features will be available in the production release of SAGEMATH by next spring.
- For the time being it won't be possible to expose HPC-GAP parallel features to SAGEMATH via its main library-level interface to GAP. On the other hand, they should be accessible to advanced users for hand-launched computations, with a separate installation of HPC-GAP and SAGE's legacy text-based interface to GAP in the near term and via `libgap` when support for HPC GAP is added by the HPC GAP developer. It could also be possible to use remote procedure calls to an HPC-GAP server which runs using the SCSCP GAP package.
- It is possible right now, with a custom build, to use the development release of PARI compiled to use multithreading, in SAGEMATH; some simple benchmarks demonstrate that SAGE benefits properly from the parallel features.
- Benefits of the the parallel finite field linear algebra library LINBOX will be directly enabled in the upcoming release 8.9 of SAGEMATH which is in beta release at the time of writing.

We note that all of these features work, or will work, so long as they are in control of the threads being allocated. However, a full-featured HPC enabled SAGEMATH could hope for

much more. In particular, we identify a number of levels of integration that could be expected in the future, each requiring progressively more effort.

- HPC components usable from SAGEMATH independently from a single PYTHON thread. User controls the number of threads.
- SAGEMATH controls the number of threads in use by the individual systems, but with the same other limitations as the first point.
- The HPC components are able to be run from libraries making use of the Python multiprocess model, with tweaks to prevent conflicts, but with much reduced performance.
- A fully-fledged HPC Sage distribution with full performance, no limitations with regard to multiprocessing and full control over the number of threads across the whole system.

Obviously the first level of integration is available in the near future as SAGEMATH begins to integrate the work completed in the OpenDreamKit project in its normal update cycle.

As we detail in the next Section, the final level of integration would likely require an unprecedented level of cooperation, expertise, planning and effort across many projects and across continents.

### 2.1. Challenges for producing a fully-fledged HPC SAGEMATH distribution

Through the expertise we have gained across OpenDreamKit and by consulting outside experts, we have compiled a list of the challenges to producing a fully-fledged HPC Sage distribution. These are in fact challenges to any sufficiently large system and especially apply to Open Source distributions which combine disparate projects developed across a wide range of communities.

In the points we identify, we refer to both process level parallelism and thread level parallelism. Both come with advantages and disadvantages. But the most important point in either case is that the components of the system need to agree on a common strategy for parallelism, including a strategy for how the two kinds of parallelism should cooperate. This is the overarching obstacle to producing an HPC Sage, because the individual components have not been designed, a priori, with such cooperation in mind from the ground up.

By "strategy" here, we mean how one should write parallel code across the whole system, how threads and processes should coordinate and how memory should be cooperatively managed.

Threading components cooperatively presents many technical challenges, especially in a heterogeneous system. In particular, components need to agree on how to make code threadsafe, how to control the number of threads and how to share memory when threads are started in the same process, and then this strategy needs to be rolled out across all components. Libraries that are currently not threadsafe then need to be made threadsafe, which can mean extensive reworking or use of automated tools such as those used in the parallel SINGULAR system.

A multiprocess approach may avoid some of the technical obstacles presented by threads, but then one needs to decide how to do data sharing and communication between processes.

We present a (non-exhaustive) list of some of the specific technical obstables that must be overcome to create a working, fully-fledged HPC distribution:

- Garbage collectors: these need to be thread aware and to cooperate with one another. Typically they aren't and they don't. It's not sufficient for each system to have threaded garbage collection (gc), as allocations in one system can trigger allocations in another, e.g. with recursive structures from one system built on structures from another system. If the gc of one system is not aware of the threads of the other system, this will lead to crashes.
- Data sharing: e.g. a GMP integer created in one system can't be handed to another system and freed or even used there because bit level gc flags prepended to the data will be different and different allocators work in different ways. This means data typically has to be copied everywhere even at a very low level, causing very high overhead when transferring information from one system to another.

- Initialisation: some packages and libraries need special per thread initialisation. For example, HPC GAP requires per thread initialisation of interpreter state. This works well in a system where GAP is controlling the whole system, but if another part of the system needs to start up threads, it has to hook into the library initialisation of HPC GAP. This is difficult, technical work and will only interoperate with other systems that have been specifically designed to cooperate with it.

  This problem is particularly acute when using packages as libraries, e.g. via libGap.
- Threaded allocators: although the system malloc is threadsafe, it is not efficient in a multithreaded environment, especially on OSX. Allocators such as Singular's omalloc are not threadsafe at all and there is a uniform slowdown of 2x using the system allocator, across the whole system. High performance threaded allocators such as tcmalloc don't alleviate this problem, and in the worst case make it even worse. This is because custom allocators are heavily tuned for the systems using them. Writing parallel custom allocators is extremely demanding work, which can take years of effort. Without them, large slowdowns may occur when running an HPC aware system in a single core environment.
- Thread safety: many libraries are not threadsafe, and making them threadsafe can be a major undertaking, for example eliminating global variables, or putting mutexes on them, linking against threadsafe allocators, having functions not modify their inputs, etc. Many libraries are currently written without these requirements in place, meaning they would have to be rewritten from scratch on any system that makes calls out to them from another threaded library. Otherwise, parts of the system would have to be turned off in a threaded environment.

  This is not just a theoretical problem. Singular for example can make use of coefficient rings provided by other systems. Parallel Gr"obner basis code may then result in the provider of those coefficient rings needing to be made threadsafe. Other packages maintain global characteristics, e.g. a modulus, which may be set and reset by different threads calling that code, resulting in inconsistent and incorrect results, which may not be detected until the code is run for the first time, or at all. In the worst case, this can result in publications having to be withdrawn because computations appeared to output meaningful results, but in fact were corrupted at run time.
- Thread control: it is very expensive on a supercomputer where time has to be paid for by core/hour if thousands of threads are started by the system because there is no uniform control to limit the number of threads. Failure to implement such a mechanism could cost users tens of thousands of dollars. We are aware of an instance where this happened to a colleague because they decided to roll their own parallel code.
- Compiler/interpreter cooperation: threaded gc usually requires cooperation and integration with the language(s) themselves. Retrofitting this kind of functionality to a language not designed to work with threaded gc is usually impossible, as languages tend to be developed by external communities and are not under the control of package developers. The work is also extremely technical and complicated.
- Different gc strategies: compacting, moving, incremental, precise, conservative, reference counting, generational. All the different collectors have different strategies, which make different assumptions, especially about object references from the stack or in CPU registers, and different assumptions about reachability, when collection can happen and what mechanisms exist to make this threadsafe. If a system requires threads to be stopped in order for part of its gc cycle (typical) then other threads can't just keep going and potentially triggering allocations during that time. Simply stopping everything for all the different allocators may lead to very poor performance across the system.

  References from one gc memory space to another need to be managed, typically with finalizers, which can be a performance bottleneck, and cycles overlapping different

gc memory spaces cannot be effectively handled, placing limitations on the types of structures that can be constructed by the system, which may place additional constraints on the mathematical user.

- Memory monitoring: many allocators more agressively clean up memory when it is running low. But in order to do this, they need reliable information about how much has been allocated. This is typically done by counting allocations across the system. But each collector needs this information and it needs to be shared across all the systems.
- Bugs: there's a saying that at scale, everything breaks. Whilst single core systems are often very tolerant of coding bugs, parallel code will expose a great number of bugs that were previously hidden. A small memory leak here, or an array overrun there will cause massive corruption or out of memory conditions or crashes in a parallel system.
- Tools: debugging multithreaded applications usually requires integrated tools that are aware of all parts of the system. It is not feasible to write and debug code that cannot effectively be debugged. It just causes massive delays in development of new features across the system, effectively bringing development to a halt. It is typical for a single bug in a multithreaded or multiprocess system to take months to resolve without such tools. Profilers similarly need to be thread/process aware and may need integration with the system to monitor it effectively.
- Language features: use of a parallel system usually requires new language elements, e.g. semaphores, mutexes, shared regions, coroutines, tasks, etc. Development of these can be prohibitive in systems designed around languages which do not have these features. For example, tasks in Gap need to cooperate with tasks in Python, otherwise neither will function.
- Heterogeneous computing: very often the resources that are available differ from site to site, or may even be heterogeneous at a given site. E.g. the code may be running on a local server, but computing resources may exist over a network on a large supercomputer. This is expected to be increasing the case in future years as Moore's law for desktop computing comes to an end. Effective use of external resources requires significant understanding of the technologies that enable this, and integration of those technologies into the system, often with language support being required.
- Platform and architecture support: often, getting high performance in parallel workloads depends on being able to natively support the operating system or architecture. Large packages like GAP and SINGULAR do not have native Windows versions. Moreover, tricks such as thread local storage can be inefficient on certain operating systems, meaning that significant rework of package code at a very technical level may be necessary to retrofit them for efficient parallel operation.

All of these obstacles show that whilst our efforts toward an HPC SAGE distribution are vital, they are just the beginning of what will be a long journey!

## REFERENCES

[1] BLUMOFE, R. D., JOERG, C. F., KUSZMAUL, B. C., LEISERSON, C. E., RANDALL, K. H., AND ZHOU, Y. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 1995), PPOPP '95, ACM, pp. 207–216.

[2] GAUTIER, T., LIMA, J. V. F., MAILLARD, N., AND RAFFIN, B. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing* (Washington, DC, USA, 2013), IPDPS '13, IEEE Computer Society, pp. 1299–1308.

[3] GAUTIER, T., ROCH, J.-L., SULTAN, Z., AND VIALLA, B. Parallel algebraic linear algebra dedicated interface. In *Proceedings of the 2015 International Workshop on Parallel Symbolic Computation* (New York, NY, USA, 2015), PASCO '15, ACM, pp. 34–43.

[4] INTEL CORPORATION. Intel threading building blocks, 2008. https://www.threadingbuildingblocks.org/.
[5] OPENMP ARCHITECTURE REVIEW BOARD. OpenMP application program interface version 5, 2018.