REPORT ON OpenDreamKit DELIVERABLE D5.13

Parallelise the Singular sparse polynomial multiplication algorithms and provide parallel versions of the Singular sparse polynomial division and GCD algorithms.

DANIEL SCHULTZ

Due on	31/08/2019 (M48)
Delivered on	30/08/2019
Lead	University of Kaiserslautern (UNIKL)

Progress on and finalization of this deliverable has been tracked publicly at:
`https://github.com/OpenDreamKit/OpenDreamKit/issues/111`

Deliverable description, as taken from Github issue #111 on 2019-08-30

- **WP5:** High Performance Mathematical Computing
- **Lead Institution:** University of Kaiserslautern
- **Due:** 2019-08-31 (month 48)
- **Nature:** Demonstrator
- **Task:** T5.4 (#102)
- **Proposal:** p. 51
- **Final report** (sources)

Singular is a computer algebra system aimed at computations in algebraic geometry and is one of the key components used by SageMath. Computing with multivariate polynomials being at the core of Singular, their performance impacts the whole system.

The aim of this deliverable was to modernize Singular's sparse multivariate arithmetic by 1) updating the algorithms to the state of the art and by 2) applying thread level parallelism to achieve decent scaling on multi-core machines. The operations we focused on are multiplication, divisibility testing, and the computation of the greatest common divisor. The implementation was carried out in the C library Flint, which is used by Singular but also by independent systems. The latter thus also benefit from the improvements. Among many other applications, this tackles the long standing slowness of multivariate rational fractions in SageMath.

Contents

1. Introduction

SINGULAR [1] represents polynomials as a linked list of terms in the sparse distributed format. For example, the polynomial $4x^2 + 5xy^2z^3 + 6yz^2$ with variables x, y, and z might be stored as

	coefficient	exponents on (x,y,z)
term 1	4	$(2, 0, 0)$
term 2	5	$(1, 2, 3)$
term 3	6	$(0, 1, 2)$

,

where each term is essentially a coefficient together with an exponent tuple. This format is optimized for Gröbner basis calculations in algebraic geometry. However, because the terms themselves are stored in a link list, the SINGULAR format is unsuitable for the arithmetic operations of multiplication, division, and greatest common divisor (GCD). For this reason SINGULAR currently relies on the library FACTORY [5] for these arithmetic operations. FACTORY is a self-contained C++ library for polynomial arithmetic that has been developed as part of SINGULAR. Since the recursive representation in FACTORY is also not particularly well suited to parallelization, we have implemented for this deliverable SINGULAR's original sparse distributed format in the library FLINT [6]. FLINT is a C library implementing basic arithmetic operations over a variety of coefficient domains and is already in direct use by SAGE for rational matrices and univariate polynomials. The implementation in FLINT uses arrays, which means that the user has random access to the terms of their polynomials, and this is crucial for our parallel arithmetic operations.

The inclusion of FLINT multivariate polynomial arithmetic has improved the single core performance of SINGULAR on our set of benchmark problems by one to several orders of magnitude. SINGULAR is one of the software components of SAGE used for multivariate arithmetic, so the users of SAGE will benefit seamlessly when the SINGULAR version is updated. Since the FLINT library itself is useful outside of SINGULAR, we present the timings for the basic arithmetic operation in FLINT as well as the timing of the operation in the new version of SINGULAR, which includes all conversion and clean-up costs associated with SINGULAR. As SINGULAR's main usage is as a Gröbner basis engine, it does not make sense to try to rewrite the polynomial format used natively by SINGULAR. Instead, the conversion cost should be viewed simply as the time needed to convert polynomials to formats optimized for different purposes.

2. Details of the systems

Besides the three monomial orders *lexicographic*, *graded lexicographic* and *graded reverse lexicographic* used commonly in SINGULAR, FLINT supports polynomials with exponents of unlimited size. Since SINGULAR has a fixed and limited size on the exponents this second feature is somewhat moot for SINGULAR users. About two years into this project we had to redesign the fundamentals of the multivariate polynomials in FLINT to achieve the desired flexibility and performance. By the end of the next two years more than $100,000$ additional lines of code dedicated to multivariate arithmetic had been added. This includes many redesigns as bottlenecks were discovered and implementations were redone.

We run our benchmarks on the server `nenepapa`, which has two sixteen core Intel Xeon E5-2697A v4 processors at 2.6 GHz and 700GB of memory. We show the timing of the basic operation in FLINT in the column labelled *flint* and the timing of the new version of SINGULAR in the column *sing*. At the time of running these benchmarks, `nenepapa` was also running two instances of long running calculations. This seemed to only slightly negatively affect the timings on 32 threads.

The largest characteristic p supported by SINGULAR for arithmetic over finite fields is $p = 2^{29} - 3$, and this is the prime we use to test arithmetic over $\mathbb{Z}/p\mathbb{Z}$. Both FLINT and SINGULAR use the GMP library for elements of $\mathbb{Z}$ with a special representation for integers less than 2^{62} (2^{61} for SINGULAR) in absolute value; small integers and elements of $\mathbb{Z}/p\mathbb{Z}$ both take one word of memory while large integers are managed by GMP. All times are reported in seconds.

Since these benchmarks deal with polynomials whose sizes are comparable to the total running time of the calculation, it is necessary to parallelize the conversion between FLINT and SINGULAR. This is a rather disappointing task as one direction is limited by the scaling of `malloc` and the other direction is limited by SINGULAR's inherently serial data structure; the time to simply traverse SINGULAR's link list can be comparable to the time to do the threaded calculation in FLINT. We encountered several performance quirks of `malloc` on `nenepapa`, which is running Gentoo Linux. The most noticeable of these was that, when constructing polynomials in SINGULAR, the throughput of the `malloc` provided by the system only starts to scale past 3 or 4 threads. Other implementations of `malloc` such as `tcmalloc` did not have this quirk but had overall higher times on 16 threads. Therefore, we simply ran all of our benchmarks with the system's default `malloc`. In order to use parallel conversion routines, the default allocator `omalloc` of SINGULAR must be disabled with the configuration option `--disable-omalloc` as `omalloc` is a special-purpose allocator that is not thread safe. Since this slows down the rest of SINGULAR by about a factor of two, it may not be advantageous to disable `omalloc` in practice. Nevertheless, we have disabled this to test the efficiency of the parallel conversion routines.

We defined the efficiency on n threads as

$$\text{efficiency} = \frac{\text{FLINT time on } 1 \text{ thread}}{n \cdot \text{FLINT time on } n \text{ threads}}.$$

In order to measure efficiency of the code, rather than the server CPU's, it is necessary to limit all CPU's to the same frequency, by disabling Intel's turbo boost, which otherwise runs the CPU at a higher frequency if fewer threads are being used (see Section 9). It is also necessary to pin threads to cores as `nenepapa` is unable to consistently schedule threads on the same core and prefers hyperthreads over physical CPU's.

3. Sparse Multiplication

Parallel multiplication has been investigated previously in [8] and [2]. The more effective strategy for sparse polynomials is in the latter and seems to be the approach of directly calculating independent pieces of the answer. This makes the algorithm essentially lock-free, while the approach of [8] requires a lock on its parallel merge. To test the effectiveness of this strategy, we time the multiplication $\cdot$ in

$$(1 + x + y + 2z^2 + 3t^3 + 5u^5)^m \cdot (1 + u + t + 2z^2 + 3y^3 + 5x^5)^n$$

for $m = n = 16$, where the product is already quite large with 28 million terms. As shown in [8], it is difficult to obtain a good speed up on this example. The reason for this is that the inputs each have only 20 thousand terms, so the majority of the time is spent writing down the answer, where only 14 additions are done per term on average. Table 1 shows the timings with 16 threads. The poor scaling of the SINGULAR times over $\mathbb{Z}$ can be explained easily: besides testing the multiplication in FLINT, this benchmark tests the creation of large polynomials in SINGULAR, which is a task bounded by the scaling of `malloc`. In addition to having larger clean-up costs, the benchmark over $\mathbb{Z}$ puts three times as much pressure on `malloc` as it does over $\mathbb{Z}/p\mathbb{Z}$. The multiplication over $\mathbb{Z}/p\mathbb{Z}$ is overall faster and scales better. The efficiency on 16 threads is 0.86 versus an efficiency of 0.76 over $\mathbb{Z}$. This is to be expected as the memory management of elements of $\mathbb{Z}$ via GMP adds overhead. As we increase the size of the problem we can observe better scaling as shown in Table 2. Now the efficiency on 16 threads is 0.90 (0.90 for $\mathbb{Z}/p\mathbb{Z}$), and the efficiency on 32 threads is 0.76 (0.82 for $\mathbb{Z}/p\mathbb{Z}$). The efficiencies for all multiplications in this section are summarized in Figure 1.

	$\mathbb{Z}$		$\mathbb{Z}/p\mathbb{Z}$	
#th	flint	sing	flint	sing
1	10.54	22.44	9.12	11.72
2	5.55	13.69	4.84	6.67
3	3.80	11.07	3.29	6.34
4	2.95	10.11	2.50	4.70
6	2.09	7.75	1.68	3.41
8	1.60	6.99	1.26	2.74
10	1.30	6.50	1.03	2.23
12	1.09	5.95	0.86	2.04
14	0.96	5.73	0.74	1.66
16	0.86	5.19	0.66	1.50

TABLE 1. Sparse multiplication for $(m, n) = (16, 16)$.

4. DENSE MULTIPLICATION

When the input polynomials have a density past a certain threshold, it is possible to do better than algorithms based on heaps. For this reason we implemented an approach based on arrays and parallelized it. The approach is suited well to the multiplication in, for example,

$$(1 + x + y + z + t)^m \cdot (1 + x + y + z + t)^n,$$

As the inputs to the multiplication in this case each have 46 thousand terms, and the product only has 635 thousand terms, the amount of work per output term is much higher than in Section 3. Table 3 shows that the efficiency on 16 threads is 0.93 in both cases. However, as this approach breaks up the input problem into a limited number of pieces, and only some of these pieces are large, this approach is effective at low thread counts but does not scale past 16 threads.

5. SPARSE DIVISION

For this benchmark we simply divide the product in Section 3 by the divisor $(1 + u + t + 2z^2 + 3y^3 + 5x^5)^n$. It is important to note that we are in fact computing two things: (1) whether

	$\mathbb{Z}$		$\mathbb{Z}/p\mathbb{Z}$	
#th	flint	sing	flint	sing
1	120.0	162.2	53.88	64.22
2	60.0	124.8	28.13	34.68
3	41.0	85.6	18.74	29.69
4	31.4	67.3	14.31	23.87
6	21.1	44.4	9.48	15.97
8	16.1	34.3	7.26	12.72
10	13.0	29.7	5.96	11.40
12	10.9	27.1	4.99	9.91
14	9.4	25.2	4.22	9.00
16	8.3	23.6	3.76	8.14
20	6.7	21.5	3.13	7.16
24	5.6	19.6	2.65	6.37
28	4.9	18.8	2.27	5.62
32	4.9	17.9	2.04	5.12

TABLE 2. Sparse multiplication for $(m, n) = (20, 20)$.

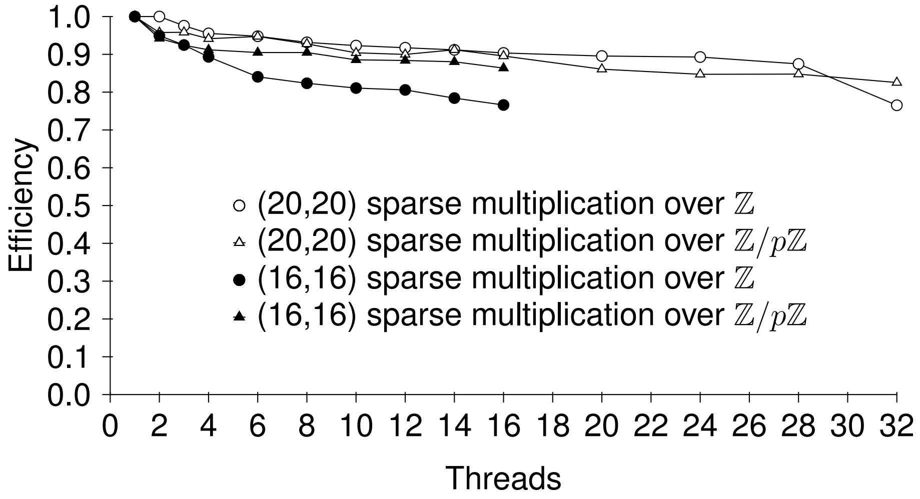

FIGURE 1. Efficiency of the sparse multiplication as defined by $\frac{\text{time on 1 thread}}{n\cdot\text{time on } n \text{ threads}}$

the dividend is divisible by the divisor and (2) the quotient if it is. As with sparse multiplication, the approach of Gastineau and Laskar [3] scales better than the approach of Monagan and Pearce [7]. Division is more difficult to parallelize than multiplication because the algorithm is highly sequential: Most terms in the quotient depend on previous terms in the quotient for their calculation. For this reason, the algorithm requires locks on the generated quotient, and only one thread can be generating quotient terms at a time. We achieve an efficiency of 0.71 on 16 threads (0.78 for $\mathbb{Z}/p\mathbb{Z}$) as shown in Table 4. Since one of the inputs to the algorithm is large, this tests not only the division in FLINT but also the conversion from SINGULAR to FLINT. In order to obtain reasonable timings with SINGULAR over $\mathbb{Z}$, it was necessary to force FLINT to borrow SINGULAR's integers. With this optimization the overhead over $\mathbb{Z}$ is much less than the corresponding overhead in Table 1. However, conversion overhead does not scale well for the following reason: The time to merely traverse SINGULAR's linked list representation of the dividend is about 0.7 seconds in this benchmark. This operation is necessary to find

	$\mathbb{Z}$		$\mathbb{Z}/p\mathbb{Z}$	
#th	flint	sing	flint	sing
1	5.08	5.39	3.64	3.71
2	2.56	2.80	1.83	1.90
3	1.71	1.90	1.22	1.28
4	1.29	1.45	0.92	0.98
6	0.86	1.00	0.62	0.66
8	0.67	0.77	0.46	0.50
10	0.52	0.63	0.37	0.40
12	0.46	0.55	0.31	0.34
14	0.40	0.50	0.28	0.30
16	0.34	0.43	0.24	0.26

TABLE 3. Dense multiplication for $(m, n) = (30, 30)$.

the polynomial's length, is an inherently serial operation, and consumes all of the conversion overhead over $\mathbb{Z}/p\mathbb{Z}$ on 16 threads.

	$\mathbb{Z}$		$\mathbb{Z}/p\mathbb{Z}$	
#th	flint	sing	flint	sing
1	9.96	13.29	9.60	11.44
2	5.22	7.48	4.74	5.82
3	3.64	5.75	3.31	4.15
4	2.68	4.66	2.54	3.17
6	1.92	3.50	1.68	2.80
8	1.55	2.93	1.34	2.06
10	1.26	2.76	1.17	1.82
12	1.14	2.54	1.03	1.57
14	0.92	2.30	0.90	1.44
16	0.88	2.01	0.78	1.30

TABLE 4. Sparse division for $(m, n) = (16, 16)$.

6. SPARSE GCD

For this benchmark we calculate $\gcd(a^{m_1}b^{n_1}, a^{m_2}b^{n_2})$, where $a = 1 + x + y^5 + z^4 + t^{40} + u^{50}$ and $b = 1 + x^9 + y^2 + z^{11} + t^7 + u^{27}$. This calculation requires at least a dozen steps to be completed in serial, and we achieve an efficiency of 0.72 on 16 threads (0.66 for $\mathbb{Z}/p\mathbb{Z}$) by parallelizing the majority of these steps as shown in Table 5. The overhead from converting between the SINGULAR format is negligible here. The algorithm over $\mathbb{Z}/p\mathbb{Z}$ suffers because, while the input problem can be split up into several pieces of work, the recombination of the results from each thread is an extra step not present in the serial algorithm. Furthermore, this recombination becomes less efficient with greater numbers of smaller pieces.

7. DISSEMINATION AND IMPACT

This project has been the topic of an extensive blog `http://wbhart.blogspot.com/2019/08/parallel-multivariate-arithmetic-final.html`. The linked article was read by over 200 individuals and was noticed by all of the leading experts in parallel polynomial arithmetic, whom we have been in constant contact with.

	$\mathbb{Z}$	$\mathbb{Z}/p\mathbb{Z}$
#th	flint	flint
1	23.24	117.8
2	12.26	59.8
3	8.24	42.5
4	6.26	32.2
6	4.41	24.2
8	3.42	17.6
10	2.87	15.6
12	2.50	13.4
14	2.23	11.3
16	2.03	11.2

TABLE 5. Sparse GCD for $(m_1, n_1) = (8, 5)$, $(m_2, n_2) = (3, 9)$.

A Jupyter notebook demonstrating our code running is available at `https://github.com/tthsqe12/SingularParallelArithmetic`.

It is worth pointing out some of the ways in which we and others have and will benefit from this work, and what are some of the hard restrictions.

There are four main areas where fast polynomial arithmetic can be expected to speed up important research applications using Singular. We will discuss each in turn.

(1) Basic arithmetic for arithmetic's sake.
(2) Gröbner bases.
(3) Primary decomposition/factorisation.
(4) Rational functions.

No. 1. Here we are talking about users implementing algorithms in Singular which need to do basic arithmetic, i.e. multiplication, division and gcd of polynomials. As the default polynomial format in Singular is the linked-list sparse distributed format, a conversion to and from Flint array sparse distributed format is unavoidable. One still gets a big speedup in practice, but as you can see from timings, conversion costs may actually dominate, meaning you don't get a linear speedup with cores. This is essential behaviour and we have expended much effort in minimizing the cost. The user still wins, however, so it is worth the effort. The timings above directly demonstrate these benefits.

No. 2. Here you can only expect to gain when there are large divisions done in the Buchberger algorithm for Gröbner bases, which is not used for every Gröbner basis application. However, it does still have significant applications. We have encountered such examples in real world applications recently (albeit with orderings that we nearly but don't quite support yet). Many such examples exist in real research with orderings that we do support and we expect big gains here. Work under this heading (supporting additional orderings, etc.) will go on for years after ODK and we will be leveraging the new multivariate engine in many new ways.

No. 3. Primary decomposition depends heavily on factorisation of multivariate polynomials. But factorisation can be a life's work, compared to a four year ODK project, and so lies completely outside the scope of ODK. Daniel Schultz has already begun work on integrating the ODK parallel multivariate work in the Singular factoring engine, as a means of disseminating our work, and the preliminary results are extremely encouraging.

No. 4. William Hart and Hans Schönemann have been implementing fast rational functions based on the ODK work. Here conversion costs do NOT occur. Rational functions are constructed in the Flint array format and never leave that format. One can even do Gröbner bases over rational function fields without conversion, and this is a very important application in Singular. However, one should temper one's expectations. Although the single core gains may make orders of magnitude difference here, it is common for the rational functions to be too small to benefit from parallelisation. However, when ”coefficient explosion” does occur, then it becomes critical. However, one should also realise that in such cases it is often better to use a modular Gröbner basis algorithm, if available. On the other hand, this may have been merely due to the prior lack of fast rational functions in the past, and even with the modular algorithm, one will still critically benefit from the ODK work, since one still has to do the same computations mod p, which our ODK work implements. The code for fast rational functions is currently essentially finished (it has been written by William Hart, independently of ODK funding but directly based on the fast ODK arithmetic). It needs some new Singular interpreter features to be added by Hans Schönemann when he returns from holidays in order to be functional. Experience tells us the speedups could potentially be orders of magnitude (seconds compared to weeks in some real world cases).

We should also mention that the new computer algebra system Oscar is already benefiting from the ODK work. For example, Oscar depends on the ODK work entirely for an important application from group theory (again using fast rational functions). There are also already applications in number theory within our research group in Kaiserslautern. That's all completely independent of ODK, but it's worth knowing that the benefits are being multiplied across multiple systems.

From an Oscar perspective, the conversion costs mentioned above are only incurred when converting *from* the Flint format *to* Singular format when a Groebner basis computation is needed, where the conversion cost is usually not relevant compared to the cost of the Groebner basis computation, which can be doubly exponential (in the worst case). Other systems are free to approach things in a similar way, and then the cost of conversion becomes a moot point. We have in fact learned a lot about how such a system should be constructed from this work, which we look forward to sharing with the community, both through our ODK dissemination and ongoing research projects.

8. Comparisons with other systems

8.1. Giac

GIAC [9] is a computer algebra kernel used in many well-known symbolic systems and calculators. At the time of writing, we were unable to install GIAC on our Gentoo server with all its optimizations enabled. Updates from the author will be provided on the aforementioned blog.

8.2. Trip

TRIP [4] is a system dedicated to computations in celestial mechanics and offers parallel polynomial multiplication over $\mathbb{Z}$ in a variety of polynomial formats. We chose the format that seemed to give the best timings on `nenepapa`. We noted that version 1.6.42 of TRIP suffered from poor scalability. One of the authors pointed out that this is due to usage of the system `malloc` and provided us with a patched version of TRIP using `jemalloc` and instructions to test it over $\mathbb{Z}/p\mathbb{Z}$. It is interesting to note that TRIP reaches an efficiency of 0.82 on our largest benchmark for *both* $\mathbb{Z}$ and $\mathbb{Z}/p\mathbb{Z}$, suggesting that it handles elements of $\mathbb{Z}$ with slightly better scaling than FLINT.

8.3. Maple

The parallel multiplication of Monagan and Pearce [8] is accessible through MAPLE. With the exception of the benchmark in Section 4, unpredictable garbage collection dominated the timings, which did not scale with the number of cores. We did not pin threads nor did we try to control the turbo setting for this machine as it did not belong to us, and only 12 threads were reliably available. Table 7 shows a perfect efficiency on 12 threads for MAPLE, but it is not the

	TRIP			TRIP patched					
	dense	sparse		dense		sparse			
#th	$(30,30)$	$(16,16)$	$(20,20)$	$(30,30)$		$(16,16)$		$(20,20)$	
	$\mathbb{Z}$	$\mathbb{Z}$	$\mathbb{Z}$	$\mathbb{Z}$	$\mathbb{Z}/p\mathbb{Z}$	$\mathbb{Z}$	$\mathbb{Z}/p\mathbb{Z}$	$\mathbb{Z}$	$\mathbb{Z}/p\mathbb{Z}$
1	24.40	21.90	130.00	25.61	30.55	24.01	27.94	140.10	175.10
2	12.30	11.50	67.2	12.78	15.31	11.92	14.01	69.86	87.47
3	8.30	8.02	45.7	8.55	10.21	8.06	9.38	46.74	59.11
4	6.21	6.02	33.4	6.44	7.65	6.13	7.18	35.24	48.51
6	4.24	3.96	22.9	4.38	5.21	4.19	5.14	25.02	31.78
8	3.17	3.29	17.3	3.36	3.97	3.19	3.80	18.70	23.31
10	2.87	2.68	15.5	2.81	3.17	2.60	3.02	15.06	18.97
12	2.47	2.30	12.9	2.34	2.86	2.19	2.60	12.62	15.60
14	2.01	2.07	11.0	1.98	2.50	1.99	2.25	10.80	13.55
16	1.78	1.96	10.1	1.79	2.08	1.96	2.00	9.72	11.67
20			9.4	1.53	1.80	1.42	1.70	7.98	9.72
24			9.0	1.36	1.52	1.30	1.44	6.72	8.50
28			8.9	1.12	1.35	1.17	1.29	6.01	7.35
32			8.6	1.016	1.075	1.10	1.21	5.36	6.79

TABLE 6. Multiplication over $\mathbb{Z}$ and $\mathbb{Z}/p\mathbb{Z}$ in Trip.

fastest algorithm for these polynomials as evinced by the super-linear speed up on low thread counts.

	$\mathbb{Z}$		$\mathbb{Z}/p\mathbb{Z}$	
#th	flint	maple	flint	maple
1	5.78	30.87	4.57	30.97
2	2.94	15.12	2.33	15.00
3	2.04	9.60	1.59	9.42
4	1.58	6.93	1.23	6.97
6	1.10	4.95	0.89	4.62
8	0.88	3.58	0.66	3.55
10	0.72	2.97	0.58	2.93
12	0.62	2.52	0.50	2.47

TABLE 7. Dense multiplication for $(m,n) = (30,30)$.

9. CODE

We limit the cpu turbo with

```
likwid-setFrequencies -g performance
```

All of our FLINT code is available in the `trunk` branch at `http://github.com/wbhart/flint2`. The timings for, say, the dense benchmark over $\mathbb{Z}$ can be generated in the profile directory of FLINT via the following commands.

```
make profile MOD=fmpz_mpoly
./build/fmpz_mpoly/profile/p-mul 16 dense 30 30
```

The `spielwiese` branch of SINGULAR at `http://github.com/Singular/Sources` incorporates our improvements to arithmetic. It is important to configure SINGULAR with the option `--disable-omalloc` to enable the parallel conversion routines. The new system command `--flint-threads` will set the number of threads Flint may use from within Singular, as demonstrated in the following Singular code.

```
ring r = 0, (x,y,z,t), dp;
poly a = (1+x+y+z+t)^30;
poly b = (1+x+y+z+t)^30;
poly p;
system("--ticks-per-sec",1000);
for (i = 1; i <= 16; i++) {
    system("--flint-threads", i);
    p = 0; time1 = rtimer; p = a*b; time2 = rtimer;
    "th(" + string(i) + "): " + string(time2 - time1) + "ms";
}
```

10. Conclusion and Future work

We have successfully sped up multivariate polynomial arithmetic in Singular over the coefficient fields $\mathbb{Q}$ and $\mathbb{Z}/p\mathbb{Z}$ while providing additional speed through the use of thread level parallelism. This was accomplished through a new set of multivariate modules in the library Flint, which can easily be integrated into other systems as well. Multivariate arithmetic is not an embarrassingly parallel problem, and the fastest single core algorithms require complicated data structures with unpredictable memory usage. Our benchmarks indicate that we have not compromised single core performance and have good scaling up to 8 threads with multiplication scaling well to 16 threads or even 32 threads on large problems.

We have started to rework Singular's multivariate factorization to benefit from the ODK improvements in Flint and begun to identify future improvements to that implementation that will maximise the performance impact of the ODK work. We have also implemented a rational function coefficient domain for Singular which will use Flint polynomials directly in Singular without incurring any conversion costs.

Since Flint and Singular are the multivariate polynomial computational work horses of many computational systems, including SageMath and Oscar, all of which can be used through the Jupyter notebook interface, the work reported on here impacts a large variety of Virtual Research Environments that can be built from the toolkit supported by OpenDreamKit.

References

[1] Wolfram Decker et al. Singular *4-1-2 — A computer algebra system for polynomial computations*. `http://www.singular.uni-kl.de`. 2019.

[2] Mickaël Gastineau and Jacques Laskar. "Highly Scalable Multiplication for Distributed Sparse Multivariate Polynomials on Many-Core Systems". In: *Proceedings of the 15th International Workshop on Computer Algebra in Scientific Computing - Volume 8136*. CASC 2013. Berlin, Germany: Springer-Verlag, 2013, pp. 100–115. ISBN: 978-3-319-02296-3. DOI: `10.1007/978-3-319-02297-0_8`. URL: `https://doi.org/10.1007/978-3-319-02297-0_8`.

[3] Mickaël Gastineau and Jacques Laskar. "Parallel Sparse Multivariate Polynomial Division". In: *Proceedings of the 2015 International Workshop on Parallel Symbolic Computation*. PASCO '15. Bath, United Kingdom: ACM, 2015, pp. 25–33. ISBN: 978-1-4503-3599-7. DOI: `10.1145/2790282.2790285`. URL: `http://doi.acm.org/10.1145/2790282.2790285`.

[4] Mickaël Gastineau and Jacques Laskar. “TRIP: A Computer Algebra System Dedicated to Celestial Mechanics and Perturbation Series”. In: *ACM Commun. Comput. Algebra* 44.3/4 (Jan. 2011), pp. 194–197. ISSN: 1932-2240. DOI: 10.1145/1940475.1940518. URL: http://doi.acm.org/10.1145/1940475.1940518.

[5] Gert-Martin Greuel, Rüdiger Stobbe, and Martine Lee. FACTORY *a C++ class library that implements a recursive representation of multivariate polynomial data*. www.singular.uni-kl.de/Manual/latest/sing_1.htm.

[6] William Hart. “Fast Library for Number Theory: An Introduction”. In: *Proceedings of the Third International Congress on Mathematical Software*. ICMS’10. http://flintlib.org. Kobe, Japan: Springer-Verlag, 2010, pp. 88–91.

[7] Michael Monagan and Roman Pearce. “Parallel Sparse Polynomial Division Using Heaps”. In: *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*. PASCO ’10. Grenoble, France: ACM, 2010, pp. 105–111. ISBN: 978-1-4503-0067-4. DOI: 10.1145/1837210.1837227. URL: http://doi.acm.org/10.1145/1837210.1837227.

[8] Michael Monagan and Roman Pearce. “Parallel Sparse Polynomial Multiplication Using Heaps”. In: *Proceedings of the 2009 International Symposium on Symbolic and Algebraic Computation*. ISSAC ’09. Seoul, Republic of Korea: ACM, 2009, pp. 263–270. ISBN: 978-1-60558-609-0. DOI: 10.1145/1576702.1576739. URL: http://doi.acm.org/10.1145/1576702.1576739.

[9] Bernard Parisse and Renée De Graeve. *Giac/Xcas version 1.5.0*. http://www-fourier.univ-grenoble-alpes.fr/~parisse/giac.html. 2018.

Disclaimer: this report, together with its annexes and the reports for the earlier deliverables, is self contained for auditing and reviewing purposes. Hyperlinks to external resources are meant as a convenience for casual readers wishing to follow our progress; such links have been checked for correctness at the time of submission of the deliverable, but there is no guarantee implied that they will remain valid.