Integrating High-Performance Polynomial Arithmetic into MAPLE

Xin Li and Wei Pan

30 January 2009

With the need of achieving high performance on practical problems, computer algebra systems devote more and more effort on implementation techniques. This phenomenon is certainly accentuated by the democratization of parallel architectures (multicores) and other hardware accelerating technologies.

In this report, we describe the integration of a C library offering highly optimized routines for symbolic computation with polynomials, into the computer algebra system MAPLE. By bridging these low-level and architecture-aware routines to high-level mathematical code, large speed-up factors are obtained for commands solving systems of non-linear equations. Moreover, the MAPLE end-user can rely on new powerful routines for implementing algebraic packages.

1 Background

1.1 Computer algebra software and high-performance computing

The computations performed by computer algebra systems (CAS) are mainly symbolic. As a result, they suffer from the well-phenomenon of *expression swell* that each of us can experiment when solving systems of equations by hand. In addition, computer algebra algorithms are often computationally intensive, not to say hard. For instance, solving a polynomial system in n unknowns and where each polynomial has degree d requires "at least" d^n operations, simply because such a system may have d^n solutions. This complexity barrier put the designers of CAS to challenge. In fact, naive implementation of computer algebra algorithms can often limit the applicability of CAS to problems that experimented scientists and engineers can solve by hand anyway.

Different tricks can be applied to allow better performances in symbolic computation. Among the most famous techniques are the so-called *modular methods* which reduce computations (say solving systems of linear or non-linear equations) with integer coefficients to computations modulo prime numbers. Either the *Chinese Remaindering Algorithm* or *Newton Iteration* are used to combine these modular calculations and obtain the desired result with integer coeffi-

cients. See [3] for details. Not only this principle allows to control expression swell but it also often reduces the complexity of computations.

Another important trick is the use of "faster algorithms" for performing fundamental operations such as polynomial or matrix multiplication. Indeed, the complexity of higher-level algorithms is often a function of that of those basic operations. Asymptotically fast algorithms for exact polynomial and matrix multiplication have been known for more than forty years. Among others, the work of Karatsuba [5], Cooley and Tukey [1], and Strassen [15] has initiated an intense activity in this area. Unfortunately, its impact on CAS has been slight until recently. One reason was, probably, the belief that these algorithms were of very limited practical interest. In [4] p. 132, referring to [12], the authors state that the FFT-based univariate polynomial multiplication is "better than the classical method approximately when $n + m \geq 600$", where n and m are the degrees of the input polynomials. In [6] p. 501, quoting Brent, Knuth writes "He (R. P. Brent) estimated that Strassen's scheme would not begin to excel over Winograd's until $n \approx 250$ and such enormous matrices rarely occur in practice unless they are very sparse, when other techniques apply." In [14], Shoup announced that his implementation of FFT-based univariate multiplication could outperform classical multiplication methods from input degrees $n, m \geq 32$. Therefore, the question became how to put into practice asymptotically fast polynomial arithmetic algorithms, since there was no doubt anymore that they could help. This question is addressed in [2] where the authors discuss how those algorithms could be made available in high-level programming language environment.

High-performance computing applied to CAS involve not only the use of "faster algorithms" but also the use of "finer implementation techniques". The success of Shoup and his followers is based on a sharp understanding of the targeted architecture and its features such as a memory hierarchy. It is well-recognized today that, for those fundamental arithmetic operations, both cache complexity and algebraic complexity are two important efficiency measures. With the move toward parallel computing, the parallel efficiency (that is, the speed-up factor divided by the number of processors) is another one.

1.2 The `modpn` library

Following the work of Shoup, our supervisor, Dr. Marc Moreno Maza, implemented asymptotically fast algorithms in the Aldor programming language. The objective was to achieve high-performance within a high-level language. The study reported in [2] shows a relative success in this enterprise and suggests that a C implementation is needed in order to better control the machine resources (CPU, memory).

In 2004, the first author started to implement FFT-based polynomial arithmetic operations in C. A major focus was put on univariate and multivariate polynomial multiplication for about two years in order to compete with the best known packages. As illustrated by the results published in [2], not only this goal was reached but our code often outperforms the packages with similar

specifications.

In 2006, more advanced operations were considered such as polynomial simplification (i.e. normal form of a polynomial w.r.t. a set of rules). New algorithms (with best known complexity) were designed and implemented [10]. The experimental results extended the success of [2]. More recently, asymptotically fast algorithms for solving systems of equations (under some assumptions) have been worked out by the two others [9]. Once again, comparative experimentation brought results favorable to our code.

Today, the `modpn` library is the set of those highly optimized C routines for multivariate polynomials. Dense and sparse representations are supported by means of techniques based on FFT and SLP (Straight-Line Programs). The source code amounts to 36,000 lines.

1.3 The `RegularChains` library

The ALDOR library of our supervisor led to the realization of another package (this time in the MAPLE language) called the `RegularChains` library [7]. The second author and other students of Dr. Marc Moreno Maza participate to the development of this library which amounts today to 121 commands and 70,000 lines of C Code

The original motivation for the `RegularChains` library is to design new high-level algorithms for solving systems of polynomial equations symbolically and studying their solutions. Key features are: solving parametric systems, computing the real solutions of polynomial systems (including in the parametric case), performing linear algebra over non-integral domains etc. Until the work reported here, the `RegularChains` library was relying exclusively on the MAPLE built-in polynomial arithmetic, which implement classical (and non-fast) polynomial arithmetic.

1.4 A first integration experience

In previous work [2, 8, 11] we have investigated the integration of asymptotically fast arithmetic operations into the computer algebra system AXIOM. Since AXIOM is based on GNU Common Lisp (GCL), we took the following approach. We realized highly optimized implementations of our fast routines in C and made them available to the AXIOM programming environment through the kernel of GCL. Therefore, library functions written in the AXIOM high-level language could be compiled down to binary code and then linked against our C code. To observe significant speed-up factors, it was sufficient to enhance existing AXIOM polynomial types with our fast routines (for univariate multiplication, division, GCD etc.) and call them in existing generic packages (for instance, for univariate square-free factorization). See [11] for details.

2 Integration into MAPLE

In this section, we discuss the integration into MAPLE of our fast arithmetic operations implemented in the `modpn` library. Most of MAPLE library functions are high-level interpreted code. This is the case for those of the `RegularChains` library and we shall illustrate in Section 3 how they could benefit from `modpn`.

2.1 Main challenges

With respect to our work with the computer algebra system AXIOM, see Section 1.4, this MAPLE integration was made more difficult by the following factors. First, compiled C code and MAPLE interpreted code are executed by two different runtime systems. This leads to data conversion, see Section 2.2.

Secondly, end-user objects must be allocated and de-allocated by MAPLE, which implies that most data conversions between C and MAPLE must be performed on the MAPLE side. (Clearly, one would prefer to implement them on the C side, as compiled and optimized code.) Thus data conversions can become major bottlenecks, see Section 2.4.

The fact that the MAPLE language does not enforce "modular programming" or "generic programming" is a third disadvantage compared to our AXIOM integration. Providing a MAPLE *connection-package* capable of calling our efficient C routines will not be sufficient to speed-up all MAPLE libraries using polynomial arithmetic. Clearly, high-level MAPLE code needs to be rewritten to call this connection-package and obtain improved performances. This is one of the goals of `FastArithmeticTools`, a new module of the `RegularChains` library discussed in Section 2.3.

Finally, the port of our C code to the different platforms that MAPLE support is discussed in Section 2.5.

2.2 Interface architecture

The high performance of the C code of `modpn` relies on its polynomial data-structures and the basic functions operating on them. For instance, while performing operations on multi-dimensional arrays (such as multi-dimensional FFTs), we transpose the data in order to optimize cache locality.

Two polynomial representations are used: multi-dimensional arrays for FFT-based computations and direct acyclic graphs (DAGs) for SLP-based computations. By default, MAPLE polynomials are encoded with DAGs too. In addition, MAPLE also makes use of a recursive dense representations, called `recden`.

We have implemented at the MAPLE level an interface between our C code and MAPLE. This interface, also called `modpn`, performs conversions between MAPLE-level and C-level polynomials. In addition, this interface wraps our C functions such that a MAPLE user can call them in a transparent manner.

The implementation of this interface uses, in fact, five polynomial encodings shown in Figure 1. The *Maple-Dag* and *Maple-Recursive-Dense* polynomials are the MAPLE built-in types mentioned above; the *C-Dag* and *C-Cube* polynomials

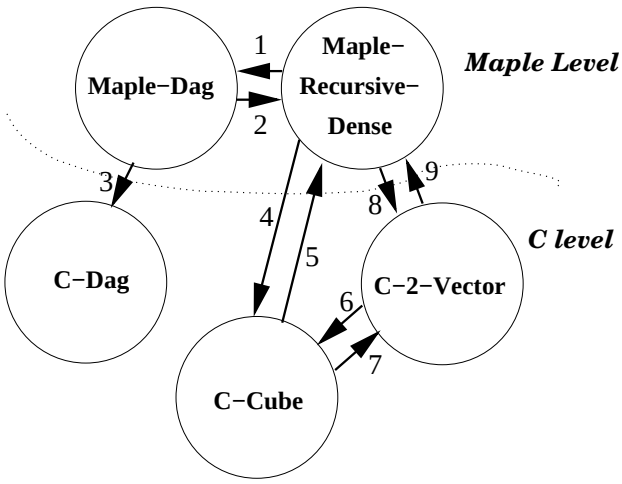

Figure 1: The polynomial data representations.

are our C encodings for SLP-based and FFT-based computations. The last type, *C-2-Vector*, has only one purpose: speed-up conversions between C polynomials and MAPLE polynomials. Indeed, the conversions between dense objects and fragmented-and-sparse objects can be relatively expensive.

2.3 The `FastArithmeticTools` module

As mentioned above, making the `modpn` library available to MAPLE users does not imply that polynomial computations in MAPLE will automatically become faster. The first reason, discussed in Section 2.2, is that `modpn` and MAPLE polynomials are encoded differently. Even after implementing the necessary data conversions, one still needs to invoke these conversions. In addition, `modpn` polynomials work modulo prime numbers. Therefore, it is necessary to use them through modular methods such that computations with integer coefficients could take advantage of them.

All these reasons have led to the development of `FastArithmeticTools`, a new module of the `RegularChains` library where we have re-implemented core operations of this library, such as *polynomials GCDs modulo a regular chain* or *iterated resultant of a polynomial modulo a regular chain*. For these operations, we have transformed the original algorithms of [13] in order to create opportunities for using fast polynomial arithmetic through modular methods. We report on experimental results in Sections 2.4 and 3.

2.4 Dealing with conversion overheads

The frequency of conversions is application dependent; it turns out that it can happen quite often in our algorithms for solving polynomial systems. Hence,

we try to reuse C objects as much as possible. Many conversions are "voluntary": we are willing to conduct them, expecting that better algorithms or better implementations can then be used in C. However, some conversions are "involuntary". Indeed, even if we would like all computationally intensive operations be carried out at the C level, our algorithms are complex, so that it becomes unrealistic to implement everything in C. Thus, there are cases where we have to convert polynomials from C to MAPLE and use its library operations.

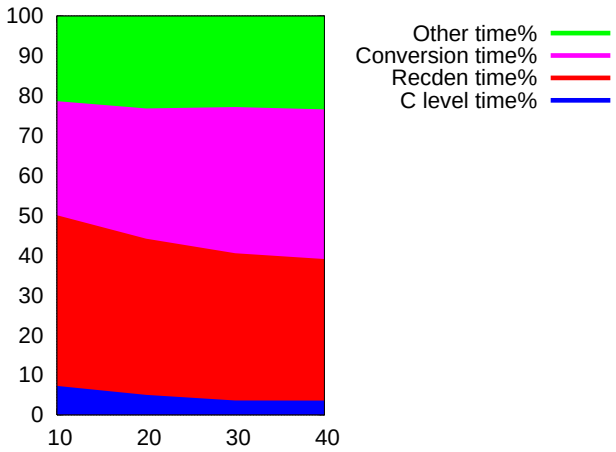

Figure 2: Bivariate solver: profiling information.

Figure 2 illustrates how time is spent when one of polynomial system solvers of the module `FastArithmeticTools` is used. On the horizontal axis is the degree of any of the input polynomials. The MAPLE code of this solver (which is specialized in solving two equations in two variables) is relatively short, about 100 lines. From `modpn`, it makes use of its basic polynomial operations, such as polynomial GCD, evaluation and interpolation.

One can see than less than 10% of the total running time is spent at the C level, despite of the fact that all the expensive algebraic computations are performed at that level! At the MAPLE level, only operations such as memory allocation, data conversions and checking the degrees of polynomials (done by `recden`) are performed.

For some applications, we were led to "push down" to C some piece of code that was originally written at the MAPLE level in our `FastArithmeticTools` module. Figure 3 shows running times (for increasing input data size) for 3 implementations of the same operation: in pure MAPLE code, in mixed code (combining MAPLE and C) and in pure C code. In this latter case, the only computations performed at the MAPLE level are the conversions of the input data and result. This benchmarked operation is the computation of so called *iterated resultants*, here of the form $\mathrm{res}(\mathrm{res}(p, T_3, X_3), T_2, X_2)$ where p, T_3 are trivariate polynomials in X_3, X_2, X_1 and T_2 is a bivariate polynomial in X_2, X_1,

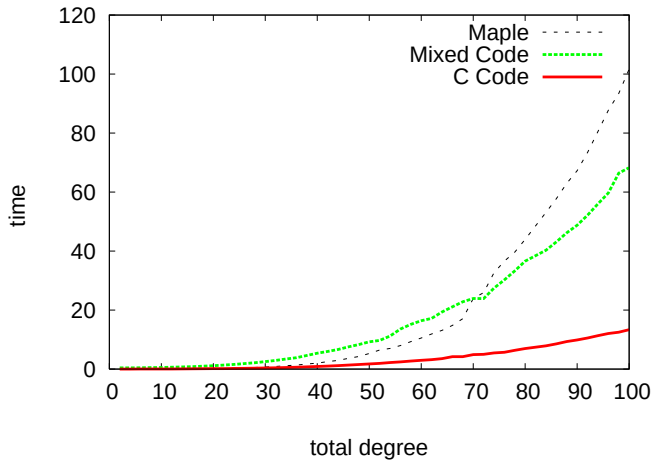

Figure 3: Invertibility test: mixed code vs MAGMA vs unmixed code.

all modulo a 27-bit prime number. Figure 3 shows that, for this operation, data conversions occuring during the intermediate computations are a bottleneck. Therefore, it was necessary to implement the whole operation in C.

2.5 Architecture port and build sequence

The integration of the C code of `modpn` has required its adaptation such that it could compile and run correctly on all the platforms that MAPLE supports. Since `modpn` was originally developed with `gcc` under Linux 32bit, this port has been relatively easy for the UNIX platforms. However, much more work was needed in the case of Windows. A last step was to include the compilation of the C-level of the `modpn` library into the build sequence of the whole MAPLE system. This part is obviously specific to MAPLE and was an interesting experience.

We are grateful to the MAPLESOFT people for their wonderful help during this integration experience.

3 Experimental Results and Outcomes

3.1 MAPLE vs MAGMA

We report here two comparative experimentations between `FastArithmeticTools` (our MAPLE code supported by the `modpn` library) and the computer algebra system MAGMA. This latter is a well-known reference for efficient implementation of polynomial arithmetic. First, we consider solving systems of bivariate polynomials. Figure 4 shows running times for our code and MAGMA on generic input systems with partial degrees d_1 and d_2. It is clear that, when the input data are large enough, our code outperforms MAGMA.

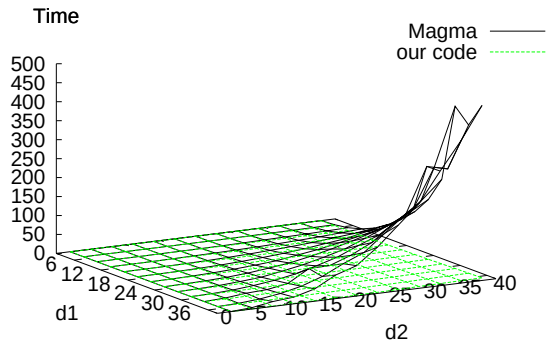

Figure 4: Generic bivariate systems: MAGMA vs. us.

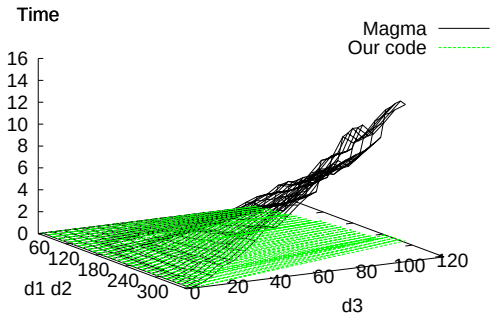

Figure 5: Normal forms, Magma vs. our code.

Secondly, we consider normal form computations. More technically, given polynomials $P(X_1), T_1(X_1), T_2(X_1, X_2), T_3(X_1, X_2, X_3)$, (where T_1, T_2, T_3 is a triangular set) such that T_i has partial degree d_j w.r.t. X_j and P has partial degree $2d_j$ w.r.t. X_j (for $1 \leq i \leq j \leq 3$) one computes the remainder (or normal form) of P w.r.t T_1, T_2, T_3. Once again, our code outperforms MAGMA.

3.2 A HPC programming tool for MAPLE end-users

We have shown that our `modpn` library provides the computer algebra system MAPLE with highly efficient routines for polynomial arithmetic. The new module `FastArithmeticTools` of the `RegularChains` illustrates that the fact that `modpn` can be used to support high-level and reach high-performance. We believe that `modpn` can become a powerful tool for MAPLE end-users.

References

[1] J. Cooley and J. Tukey. An algorithm for the machine calculation of complex Fourier +series. *Math. Comp.*, 19:297–301, 1965.

[2] A. Filatei, X. Li, M. M. Maza, and E. Schost. Implementation techniques for fast polynomial arithmetic in a high-level programming environment. In *Proc. ISSAC'06*. ACM Press, 2006. to appear.

[3] J. Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, 1999.

[4] K. Geddes, S. Czapor, and G. Labahn. *Algorithms for Computer Algebra*. Kluwer Academic Publishers, 1992.

[5] A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, (7):595–596, 1963.

[6] D. E. Knuth. *The Art of Computer Programming*, volume 2. Addison Wesley, 1999.

[7] F. Lemaire, M. Moreno Maza, and Y. Xie. The `RegularChains` library. In I. S. Kotsireas, editor, Maple Conference 2005, pages 355–368, 2005.

[8] X. Li and M. Moreno Maza. Efficient implementation of polynomial arithmetic in a multiple-level programming environment. In *ICMS'06*, pages 12–23. Springer, 2006.

[9] X. Li, M. Moreno Maza, and W. Pan. Computations modulo regular chains, 2009. Submitted to ISSAC'09.

[10] X. Li, M. Moreno Maza, and É. Schost. Fast arithmetic for triangular sets: From theory to practice. In *ISSAC'07*, pages 269–276. ACM, 2007.

[11] X. Li, M. Moreno Maza, and É. Schost. On the virtues of generic programming for symbolic computation. In *ICCS'07*, volume 4488 of *Lecture Notes in Computer Science*, pages 251–258. Springer, 2007.

[12] R. T. Moenck. Practical fast polynomial multiplication. In *SYMSAC '76: Proceedings of the third ACM symposium on Symbolic and algebraic computation*, pages 136–148, New York, NY, USA, 1976. ACM Press.

[13] M. Moreno Maza. On triangular decompositions of algebraic varieties. Technical Report TR 4/99, NAG Ltd, Oxford, UK, 1999. http://www.csd.uwo.ca/∼moreno.

[14] V. Shoup. A new polynomial factorization algorithm and its implementation. *J. Symb. Comp.*, 20(4):363–397, 1995.

[15] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik.*, 13:354–356, 1969.