Anatomy of SINGULAR

talk at

MaGiX@LIX 2011

-

Hans Schönemann

hannes@mathematik.uni-kl.de

Department of Mathematics

University of Kaiserslautern

Overview of SINGULAR

- Computations in very general rings, including polynomial rings, localizations hereof at a prime ideal and tensor products of such rings. This includes, in particular, Buchberger's and Mora's algorithm as special cases.
- Many ground fields for the above rings, such as the rational numbers, finite fields Z/p, p a prime ≤ 32003, finite fields with $q = p^n$ elements, transcendental and algebraic extensions, floating point real numbers, even rings: integers, Z/m, etc.
- Usual ideal theoretic operations, such as intersection, ideal quotient, elimination and saturation and more advanced algorithms based on free resolutions of finitely generated modules. Several combinatorial algorithms for computing dimensions, multiplicities, Hilbert series . . . .

Overview of SINGULAR II

- A programming language, which is C-like and which is quite comfortable and has the usual if-else, for, while, break . . . constructs.
- Library of procedures, written in the SINGULAR language, which are useful for many applications to mathematical problems.
- Links to communicate with other systems or with itself. Link types: Ascii, MP, ssi, SCSCP (experimental).
- can be compiled and used as a C++ library.

Algorithms in the Kernel (C/C_{++})

- Standard basis algorithms (Buchberger, SlimGB, factorizing Buchberger, FGLM, Hilbert–driven Buchberger, ...)
- Syzygies, free resolutions (Schreyer, La Scala, ...)
- Multivariate polynomial factorization
- absolute factorization (factorization over algebraically closed fields)
- Ideal theory (intersection, quotient, elimination, saturation)
- combinatorics (dimension, Hilbert polynomial, multiplicity, ...)
- many libraries: ... control.lib, surf.lib, solve.lib, primdec.lib, resolve.lib,....

Parts of Singular

- external: `GMP`: long integers, long floats
- external: `NTL`: univariate GCD, univariate factorization
- `omalloc` memory management
- `factory/libfac` multivariate GCD and factorization, etc.
- `kernel`: coefficient arithmetic, polynomial arithmetic, non-commutative rings, Gröbner bases/standard bases/syzygies, operation with ideals/free modules, linear algebra, numerical solving
- `interpreter`: flex/bison generated, calls via tables
- `SINGULAR libraries`

Problems for an efficient implementation

- How should polynomials and monomial be represented and their operations be implemented?
- What is the best way to implement coefficients?
- How should the memory management be realized?
- choosing the right algorithm (FGLM, Gröbner walk, standard basis computation driven by Hilbert function, etc.)

Monomial representations

- Macaulay 3.0 (1994): encode monomial by coefficient and an integer (enumerating all monomial by the monomial ordering) comparing is very fast, multiplication slow, divisibility test improved by a second represention for head terms: vector of exponents
  degree bound
- PoSSo (1993-1995): encode monomial by coefficient and exponent vector and ordering vector:
  (the exponent vector multiplied by the order matrix): only lexicographical comparison necessary (fast)
  fast monomial operations: simply add the complete vector for multiplication etc.
  but used a ”lot” of memory for each monomial

Monomial representations

- CoCoA: Hilbert driven algorithm (1997): bit support for fast divisibility tests
- Faugéres Algorithm F_4 (1999): monomial correspond to matrix entries: a monomial is a coefficient and a (column) number
- SINGULAR 1.4: exponent vector as char/short, operations on an array of long: smaller representation, vectorized monomial operations.
- SINGULAR 2.0: exponent vector as bit fields, operations on an array of long: smaller representation, vectorized monomial operations, Geo buckets, divisibility tests by generalized bit support
- SDMP (Maple): simplified version of the representation above

Monomial representations in SINGULAR 2-0

- bit fields for exponents
- degree of (sub-)sets of variables according to the monomial ordering

For example $9ab^2x^3y^4z \in K[a,b][x,y]$ with an degree-reverse-lex. ordering on both blocks of variables will be representetd as:
$(9,((3),(1,2)),((8),(3,4,1)))$ coefficient: 9
degree for first block (a,b): 3
exponents first block: 1,2
degree for second block (x,y,z): 8
exponents second block: 3,4,1
used space: 5 words

Bit support

use a machine int (integer $\in 0..2^{31}$ resp. 2^{63}) for an pre-test

- > 16 variables: use 1 bit per variable:
  bit i = 1: exponent of x_i is non-zero
- $10..16$ variables: use 2 bits per variable:
  field i = 00: exponent of x_i is 0
  field i = 01: exponent of x_i is 1
  field i = 11: exponent of x_i is > 1
- $9..10$ variables: use 3 bits per variable
  ...

Geo buckets

experimental implementation in Macaulay 3.0 (1998) by Yan

- lazy addition of polynomials: try to add only polynomials of the ßamelength
- store polynomials as n-tupel of partial polynomials (of length 4, $4^2, ..., 4^n$)
- extract leading term from the leading terms of the partial polynomial (if needed)
- simplify a bucket to a normal polynomial after some operations

Memory management I

Most of SINGULAR's computations boil down to primitive polynomial operations like copying, deleting, adding, and multiplying of polynomials. For example, standard bases computations over finite fields spent (on average) 90 % of their time realizing the operation `p - m*q` where `m` is a monomial, and `p,q` are polynomials.
Size of monomials: minimum size is 3 words, average size is 4 to 6 machine words

Memory management II

Requirements of a memory manager for SINGULAR:

(1) allocation/deallocation of (small) memory blocks must be extremely fast

(2) consecutive memory blocks in linked lists must have a high locality of reference

(3) the size overhead to maintain small blocks of memory must be small

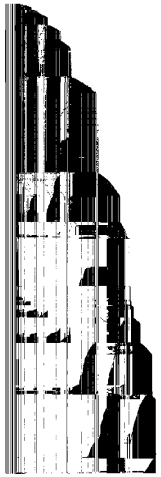
(4) the memory manager must have a clean API and it must support debugging

(5) the memory manager must be customizable, tunable, extensible and portable

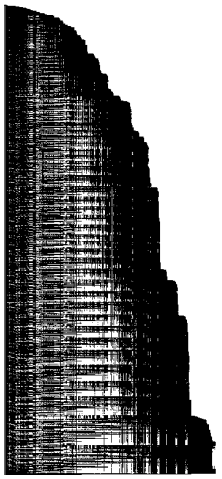
Memory management III

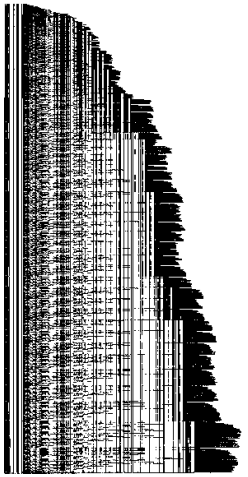
OMALLOC manages small blocks of memory on a per-page basis. That is, each used page is split up into a page-header and equally-sized memory blocks. The page-header has a size of 6 words (i.e., 24 Byte on a 32 Bit machine), and stores (among others) a pointer to the free-list and a counter of the used memory blocks of this page. On memory allocation, an appropriate page (i.e. one which has a non-empty free list of the appropriate block size) is determined based on the used memory allocation mechanism and its arguments. The counter of the page is incremented, and the provided memory block is dequeued from the free-list of the page.

- very fast allocation/deallocation of small memory blocks
- high locality of reference (may be further improved by using specific pages (i.e. specific free lists) for certain elements)
- small maintenance size overhead: 24 Bytes per page (0.6 %)



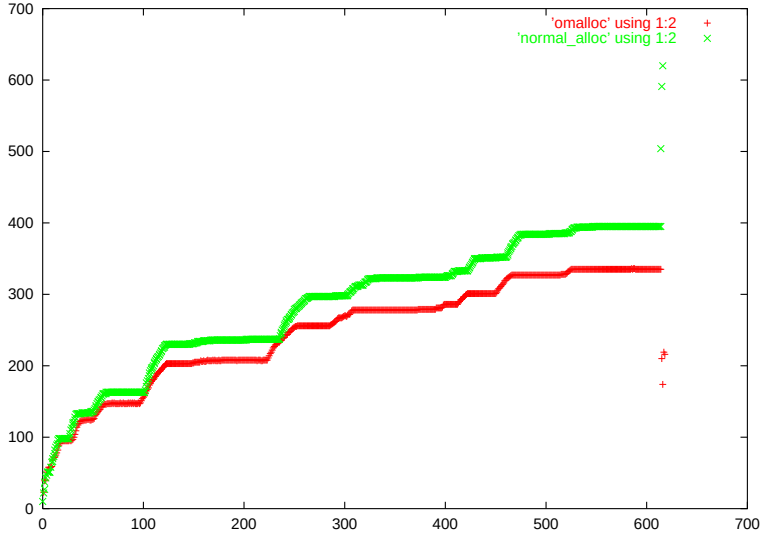
example in char p

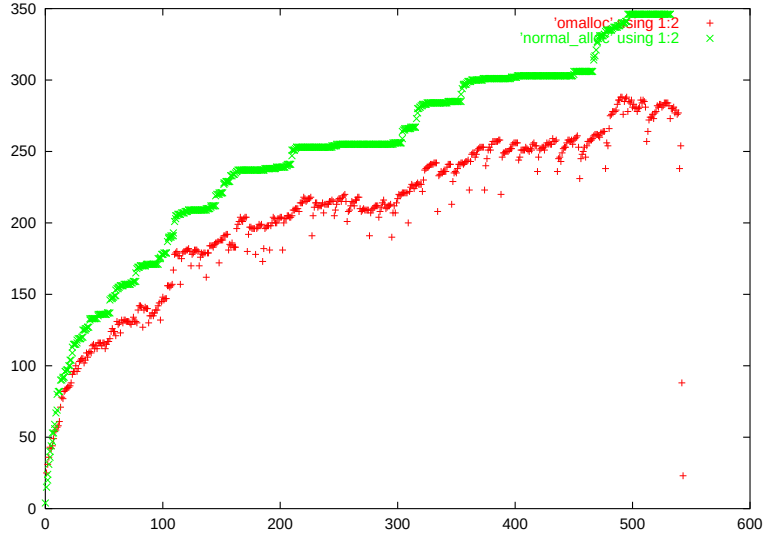


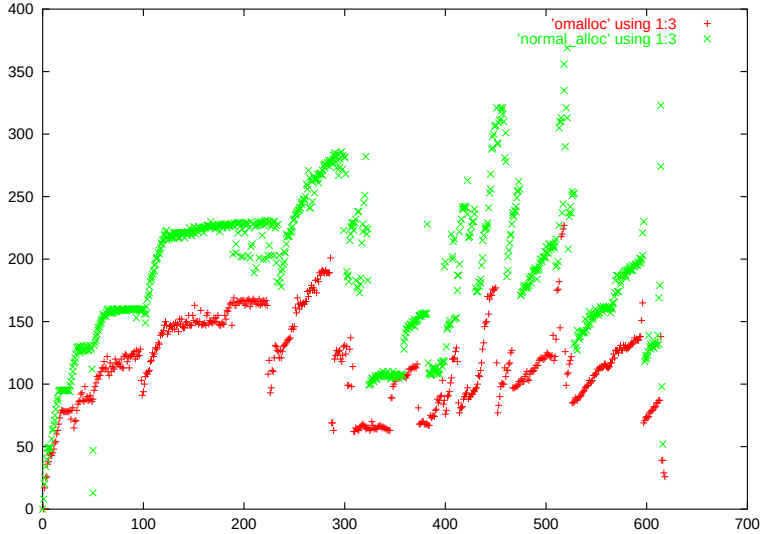


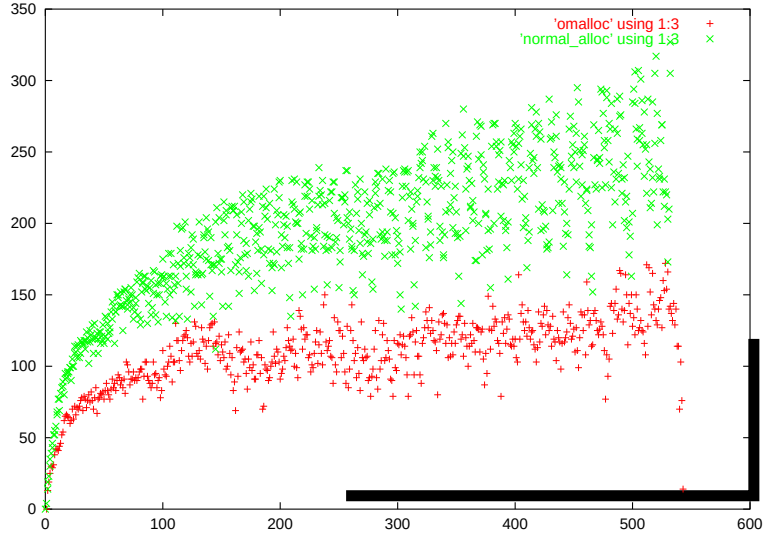
example in char 0

Allocated and active pages







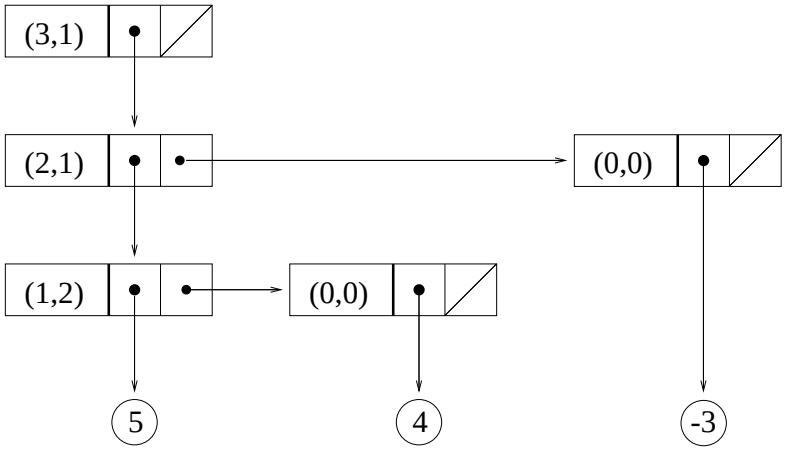


example in char p

example in double p

Encoding of polynomials: Factory

- each polynomial is represented as a univariate polynomial which has elements of a polynomial ring as coefficients.
- ordering of the variables: the level of the variable, an integer.
- the level of a polynomial is the maximum of the level of its parts
- if f.level()==0: base domain ($\mathbb{Z}$,$\mathbb{Q}$,$\mathbb{Z}/_{l}$,...)
- $0>$ f.level(): algebraic extension
- $0 <$ f.level(): (nonconstant) polynomial



coeffs/rings: separation of classes

- `number` is the type for coeffcients, `coeffs` holds additional parameters and the function table
- `poly` is the type for polynomials, `ring` holds additional parameters and the function table

Templates for polynomial operations

- a general version of each routine which uses procedures from the tables in `coeff/ring`
- more versions depending on the size of the monomial (loop unrolling), the type of the coefficients (inlining) exist
- currently: 15 routines, 2173 implementations via macro expansion

Tables for the interpeter

- (operation, argument type(s)) -> procedure to call
- automatic type conversions (type A, type B) -> procedure for conversion

Parallelization

- coarse: seperate processes on (possibly) seperate machines: via links (ssi,MP)
- fine: via threads - planned.

SINGULAR as a C/C++ library

- C++-class wrapper for `poly` etc.: in preparation
- direct use of `poly`, etc.
- use `libsingular.a`: all parts of SINGULAR in one file
- use `libsingular.so`: main part of SINGULAR (kernel, interpreter) in one file

currently used by:

- SAGE (libsingular.so)
- gfan (experimental: libsingular.a)
- gap (experimental: libsingular.so)

Usage of SINGULAR as a C/C++ library

- send a string of Singular commands to the interpreter
- call via tables in the interpreter (universal interface)
- call routines for polynomials, ideals etc.

Restructuring of SINGULAR as a set of many C/C++ libraries

- external: `GMP`: long integers, long floats
- external: `FLINT (planned)`: univariate GCD, univariate factorization
- `omalloc` memory management
- `factory/libfac` multivariate GCD and factorization, etc.
- `libcoeff` coeffcient arithmetic
- `libpoly` polynomial arithmetic (including non-commutative rings)
- `kernel` - planned as several libs: Gröbner bases/standard bases/syzygies, operation with ideals/free modules, linear algebra, numerical solving
- `interpreter`: flex/bison generated, calls via tables
- `SINGULAR libraries`

SINGULAR: www.singular.uni-kl.de