LinBox founding scope allocation, parallel building blocks, and separate compilation

Jean-Guillaume Dumas[1], Thierry Gautier[2], Clément Pernet[2], and B. David Saunders[3]

[1] Laboratoire J. Kuntzmann, Université de Grenoble. 51, rue des Mathématiques, umr CNRS 5224, bp 53X, F38041 Grenoble, France, Jean-Guillaume.Dumas@imag.fr.
[2] Laboratoire LIG, Université de Grenoble et INRIA. umr CNRS, F38330 Montbonnot, France. Clement.Pernet@imag.fr, Thierry.Gautier@inrialpes.fr.
[3] University of Delaware, Computer and Information Science Department. Newark / DE / 19716, USA. saunders@udel.edu.

1 Introduction

As a building block for a wide range of applications, computational exact linear algebra has to conciliate efficiency and genericity. The goal of the LinBox project is to address this problem in the design of an efficient general-purpose C++ open-source library for exact linear algebra over the integers, the rationals, and finite fields. Matrices can be either dense, sparse or black box (i.e. viewed as a linear operator, acting on vectors only). The library proposes a set of high level linear algebra solutions, such as the rank, the determinant, the solution of a linear system, the Smith normal form, the echelon form, the characteristic polynomial, etc. Each of these solutions involves a hybrid combination of several specialized algorithms depending on the domain, and the type of matrix. Over a finite field, the building blocks are an efficient implementation of Wiedemann and block Wiedemann algorithms combined with preconditioners [1] for black box matrices, a sparse Gaussian elimination for sparse matrices and the BLAS based dense linear algebra techniques of the FFLAS library [4] for dense matrices. The solutions over the integers and rationals are lifted from modular computations by a Chinese remainder algorithm or p-adic lifting. The design is based on high genericity to allow us to write efficient algorithms independent of the many representations of domains and matrices. As a middleware, the library relies on the efficiency of kernel libraries such as GMP[4], Givaro[4], NTL[4], ATLAS[4] and can be used by general purpose computer algebra systems such as Sage[4] or Maple[4].

We describe in this paper a selection of ideas and improvements that were recently introduced into the the design of LinBox for the forthcoming 2.0 release.

2 The lightweight founding scope allocation model

The main objects that require memory allocation in LinBox are base field or ring elements, vectors, matrices, and polynomials. The memory management for

[4] gmplib.org, www-ljk.imag.fr/CASYS/LOGICIELS/givaro, www.shoup.net/ntl, math-atlas.sourceforge.net, sagemath.org, www.maplesoft.com.

all of these object types follows the same rules, organized to maximize efficiency in time and space, and consequently requiring some efforts by the programmer. In particular no external garbage collection mechanism is used.

The input and output types of most functions are usually template types, and can be either basic types, or complicated objects. Consequently, passing arguments by value (copy) must be avoided as much as possible. Every argument is passed as a reference, including the return types. More precisely the return value of a function is also the first argument, defined as a non const reference.

```
Matrix& someFunction(Matrix& result, const XXX& args);
```

This convention was already presented in [2, §2.1] for the design of field and ring arithmetic. It does require a redefinition of the interface for some `stl`-like operators, as discussed in section 3.1. A consequence of the above convention is that the objects returned by a function, have to be declared and initialized (in particular, memory allocated, e.g. via constructors) before the function call. By enforcing this practice, we require that the programmer keep *the handle* on the objects that he allocates until all uses of the object and it's subobjects are completed. Moreover, he is responsible for object deallocation in the same scope where it was allocated. This restricts some convenient programming practices, but provides precise control of memory usage. This is particularly important when large, memory filling, matrices are in play. It also allows to avoid the costs of garbage collection or reference counting.

Many LinBox objects involve a handle containing a reference to the free store. Note that even though a function does not allocate the handle itself, it is in certain cases still free to resize and thus reallocate the free store memory referenced.

Dense Matrix allocations. The objects storing dense matrices require a special care concerning their allocations. Dense matrices are represented as a one dimensional array storing elements in the row major format: `A[i,j] = *(A+i*n+j)`. It is important to be able to define a submatrix as a view on such an array, without allocating the data. For this we propose to distinguish two classes: one for allocated (via constructors) matrices and the other for sub-matrix views. The genericity of the template mechanism or inheritance will allow to use these two types in the same code, without duplication. This allows also for an automatic decision about deallocation. Other solutions includes reference counting and explicit "end of use" functions.

3 Software abstraction layer for parallelism

Efficient parallel applications must take into consideration hardware characteristics (number of cores, memory hierarchy, etc.). It is time consuming or impossible for a single developer to program a high performance computer algebra application, with state of the art algorithms, while exploiting all the available parallelism. In order to separate the domains of expertise we have designed a software abstraction layer between computer algebra algorithms and parallel implementations which may employ automatic dynamic scheduling.

3.1 Parallel building blocks

Computer algebra algorithms have three main characteristics: 1) they are complex and require a deep knowledge of the problem in order to obtain the most efficient sequential algorithm; 2) they may be highly irregular. This enforces a runtime use of load balancing algorithms; 3) they are generic in the sense that they are usually designed to work over several algebraic domains.

In the case of LinBox algorithms, we have decided to base our software abstraction, called *Parallel Building Blocks (PBB)*, on the STL algorithms (Standard Template Like) principles. Indeed, C++ data structures in LinBox let us have random access iterators over containers which are naturally parallel. We have already defined several STL-like algorithms and the list will be extended in the near future:
for_each, transform, accumulate[5]: the PBB versions of these algorithms are similar to the STL versions except that the involved operators (or function object classes), given as parameters, are required to have their return value reference passed as the first parameter of the function. This is in accordance with the memory model of LinBox. The STL return-by-value semantic is not appropriate.

The fundamental idea of PBB is that at the computer algebra level, the parallelization of all the loops and more generally of all the STL-like algorithms will already enable good performance and easy switching among multiple implementations. Regarding performance, this parallelization of the inner loops of the underlying linear algebra is sufficient in many cases. Regarding implementations, this abstraction provides for programming independent of the parallel model with selection of the parallel environment depending on the target architecture. The parallel blocks can be implemented using many different parallel environments, such as OpenMP[6]; TBB[7] (Thread Building Blocks) or Kaapi [6]; using both static and dynamic work-stealing schedulers [8]. The current implementations are built on OpenMP and Kaapi.

3.2 Accumulate_until and early termination

To bound the complexity of many linear algebra problems, one of the key ideas is to use an accumulation with *early termination.*

For instance, this is used in Chinese Remaindering algorithms. The computation is performed modulo a sequence of (co)prime numbers and the result is built from a sequence of residues, *until* a condition is satisfied [3]. The termination of the algorithm depends on the accumulated result.

In order to parallelize such algorithms, we proposed an extension of the STL algorithms called **accumulate_until** . The algorithm takes an array v of length N, a unary operator f to be applied to each array entry and a specific binary update operator/predicate for the accumulation. This *accumulator* with a signature

[5] `www.sgi.com/tech/stl`
[6] `openmp.org`, `threadingbuildingblocks.org`

like `bool accum(a, b)` behaves like an in place addition (`a+=b`) and returns `true` to indicate sufficiently many values are accumulated. Let $S_k = \sum_{i=0,..,k} f(v[i])$ with $k \in \{0, N\}$. The algorithm computes and returns $n \leq N$ and S_n such that one accumulation during the computation of S_n returned `true` or $n = N$. In indended use, we know any additional accumulation would also return `true`.

This algorithm will be used for the early termination Chinese remaindering algorithms of LinBox. Though not yet using PBB and **accumulate_until** , a sequential version and parallel versions with OpenMP and Kaapi can be found in the LinBox distributions as `linbox/algorithms/cra-domain-*.h`.

3.3 Memory contention and thread safe allocation

Many computer algebra programs allocate dynamic memory for the intermediate computations. Several experiments with LinBox algorithms on multicore architectures have shown that these allocations are quite often the bottleneck. An analysis of the memory pattern and experiments with three well known memory allocators (ptmalloc, Hoard and TCMalloc from Google Perf. Tools[7]) have been conducted. The goal was to decide whether the parallel building blocks model was suitable to high-performance exact linear algebra. We used dynamic libraries to exchange allocators for the experiments, but one can use them together in the LinBox library if needed [7, §7]. Preliminary experiments on early terminated Chinese remaindering, not the easiest to parallelize, have demonstrated the advantage, in our setting, of TCMalloc over the others [3]. One of the main reasons for that fact is that our problems required many temporary allocations. This fits precisely the thread safe caching mechanism of TCMalloc.

4 Automated Generic Separate compilation

LinBox is developed with several kinds of genericity: 1) genericity with respect to the domain of the coefficients, 2) genericity with respect to the data structure of the matrices, 3) genericity with respect to the intermediate algorithms. While this is efficient in terms of capabilities and code reusability, there is a combinatorial explosion of combinations. Consider that each of 50 arithmetic domains may be combined with each of 50 matrix representations in each of 10 intermediate algorithm forms for a single problem as simple as matrix rank. This lengthens the compilation time and generates large executable files.

For the management of code bloat LinBox has used an "archetype mechanism" which enables, at the user's option, to switch to a compilation against abstract classes [2, §2.1]. However, this can reduce the efficiency of the library. Therefore, we propose here a way to provide a generic separate compilation. This will not deal with code bloat, but will reduce the compilation time while preserving high performance. This is useful for instance when the library is used with unspecialized calls. This is largely the case for some interface wrappers

[7] `goog-perftools.sourceforge.net/doc/tcmalloc.html`

to other Computer algebra systems such as Sage or Maple. Our idea is to automate the technique of [5] which combines compile-time instantiation and link-time instantiation, while using template instantiation instead of void pointers. The mechanism we propose is independent of the desired generic method, the candidate for separate compilation, and is explained in algorithm 1.

Algorithm 1 C++ Automatic separate compilation wrapping

```
Input: A generic function func.
Input: Template parameters TParam for separate specialization/compilation of func.
Output: A generic function calling func with separately compiled instantiations.
1: Create a header and a body files “func_instantiate.hpp” and “func_instantiate.cpp”;
2: Add a template function func_separate, with the same specification as func, to
   the header;
3: Its generic default implementation is a single line calling the original function func.
   {This enables to have a unified interface, even for non specialized class.}
4: for each separately compiled template parameter TParam do
5:    Add a non template specification funcTParam, to the header file;
6:    Add the associated body with a single line returning the instantiation of func
      on a parameter of type TParam, to the body file;
7:    Add an inline specialization body of func_separate on a parameter of type
      TParam with a single line returning funcTParam, to the header file;
8: end for
9: Compile the body file “func_instantiate.cpp”.
```

This Algorithm is illustrated on figure 1, where the function is the `rank` and the template parameter is a dense matrix over $GF(2)$, `DenseMatrix<GF2>`.

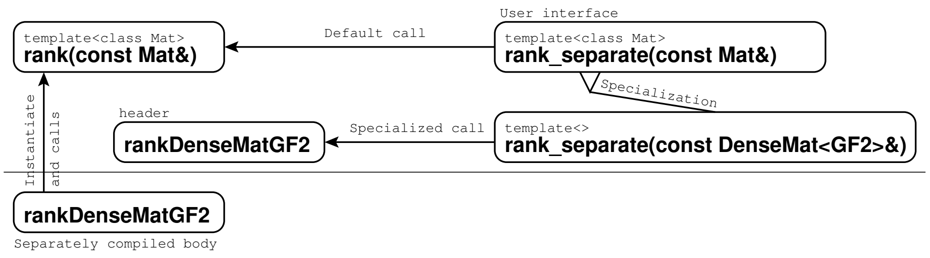

Fig. 1. Separate compilation of the rank

Algorithm 1 has been simplified for the sake of clarity. To enable a more user-friendly interface one can rename the original function and all its original specializations `func_original`; then rename also the new interface simply `func`. With the classical inline compiler optimizations, the overhead of calling `func_separate` is limited to single supplementary function call. Indeed all the one line additional methods will be automatically inlined, except, of course, the one calling the separately compiled code. If this overhead is too expensive, it suffices to enclose all the non generic specializations of “func_instantiate.hpp” by a macro test. At compile time, the decision to separately compile or not can be taken according to the definition of this macro.

We show in table 1 the gains in compilation time obtained on two examples from LinBox: the `examples/{rank,solve}.C` algorithms. Indeed, without any specification the code has to invoke several specializations depending on run-time discovered properties of the input. For instance `solve.C` requires 6 specializations for sparse matrices over the Integers or over a prime field, with a sparse elimination, or an iterative method, or a dense method, if the matrix is small...

file	real time	user time	sys. time	real time	user time	sys. time
	Rank			Solve		
`instantiate.o`	143.43s	142.47s	0.90s	171.62s	170.42s	1.12s
`{rank,solve}.o`	**18.58s**	**18.26s**	**0.30s**	**23.13s**	**22.80s**	**0.32s**
`link`	0.80s	0.64s	0.15s	0.85s	0.70s	0.14s
`Sep. comp. total`	162.81s	161.37s	1.35s	195.60s	193.92s	1.58s
`Full comp.`	162.02s	160.47s	1.21s	191.47s	189.52s	1.40s
`speed-up`	8.4	8.5	2.7	8.0	8.1	3.0s

Table 1. linbox/examples/{rank,solve}.C compilation time on an AMD Athlon 3600+, 1.9GHz, with gcc 4.5 -O2. `instantiate.o` contains to the separately compiled instantiations (e.g. densegf2rank in figure 1); `{rank,solve}.o` contains to the user interface and generic implementation compilation; `link` corresponds to the linking of both `.o` and the library; `Full comp.` corresponds to the compilation without the separate mechanism.

Acknowledgment

We thank the LinBox group and especially Brice Boyer, Pascal Giorgi, Erich Kaltofen, Dan Roche, Brian Youse for many useful discussions in particular during the recent LinBox developer meetings in Delaware and Dublin.

References

1. L. Chen, W. Eberly, E. Kaltofen, B. D. Saunders, W. J. Turner, and G. Villard. Efficient matrix preconditioners for black box linear algebra. *Linear Algebra and its Applications*, 343-344:119–146, 2002.
2. J.-G. Dumas, T. Gautier, M. Giesbrecht, P. Giorgi, B. Hovinen, E. Kaltofen, B. D. Saunders, W. J. Turner, and G. Villard. LinBox: A generic library for exact linear algebra. In A. M. Cohen, X.-S. Gao, and N. Takayama, editors, *Proceedings of the 2002 International Congress of Mathematical Software, Beijing, China*, pages 40–50. World Scientific Pub., Aug. 2002.
3. J.-G. Dumas, T. Gautier, and J.-L. Roch. Generic design of chinese remaindering schemes. In M. Moreno-Maza and J.-L. Roch, editors, *PASCO 2010*. Université de Grenoble, France, July 2010.
4. J.-G. Dumas, P. Giorgi, and C. Pernet. Dense linear algebra over word-size prime fields: the fflas and ffpack packages. *ACM Trans. Math. Softw.*, 35(3):1–42, 2008.
5. U. Erlingsson, E. Kaltofen, and D. Musser. Generic Gram-Schmidt orthogonalization by exact division. In *ISSAC'1996*, pages 275–282, July 1996.
6. T. Gautier, X. Besseron, and L. Pigeon. KAAPI: a thread scheduling runtime system for data flow computations on cluster of multi-processors. In *PASCO'07*, pages 15–23, 2007.

7. E. Kaltofen, D. Morozov, and G. Yuhasz. Generic matrix multiplication and memory management in LinBox. In *ISSAC'2005*, pages 216–223, July 2005.
8. D. Traore, J. L. Roch, N. Maillard, T. Gautier, and J. Bernard. Deque-free work-optimal parallel STL algorithms. In *EUROPAR 2008*, Las Palmas, Spain, aug 2008.