PAC++ system and parallel algebraic numbers computation

Th. Gautier and J.L. Roch
LMC, 46 av Félix Viallet, 38000 Grenoble, France
e-mail : Thierry.Gautier@imag.fr, Jean-Louis.Roch@imag.fr

Abstract: In order to manage large data structures and solve high dimension problems, computer algebra systems were re-designed in the past few years to use parallel computers. Following the evolution of the machines, our PAC system has been re-built to obtain more efficiency and scalability. This paper gives an overview of our new concepts about the integration of a computer algebra system in a parallel environnement and how the handling of the algebraic numbers can be done in a efficient way: this is illustrated by a new algorithm for computing the Jordan normal form.

Keywords: parallel algebraic computing, algebraic number, Jordan normal form, dynamic load-balancing

1 Introduction

Several parallel systems are available for algebraic computations, for both shared and distributed memory architectures. Using early versions of these environments [31, 2, 18], including our PAC (*Parallel Algebraic Computing*) library [25, 24], the programmer had to write his algorithms at a low level to manage his parallel processes and the communications between them.

Now, high level programming models and interfaces are available either on shared [9] or distributed memory machines [29, 6]. However if these solutions succeed in using "few processors", *i.e.* 1 to 32, or if they work at a coarse level of granularity with good speed-ups, it is still an open question to provide massively parallel libraries, *i.e.* targeted to parallel machines with 32 to 1024 processors, for algebraic computations. It is widely accepted that the main challenge is to distribute the work load among the processors, given that this load can greatly vary during execution. We introduced in [26, 27] an original approach to solve this problem at low levels of granularity.

In this context, the aim of this paper is to show how the PAC project has moved to PAC++, not only to take into account the requirement to dynamically distribute the work load on new massively parallel architectures, but also to provide high level facilities for the programmers. This project aims at providing an efficient and scalable library for parallel symbolic and numerical computations on distributed memory architectures. Currently, few algorithms have a parallel implementation but the global organization and all basic primitives are specified for an easy implementation of a rich parallel library.

This paper is divided in two parts. The first one begins with a presentation of the concepts and organisation of PAC++. An important point is the basic feature to communicate PAC++ objects. This point concerns *sequential* and *parallel* communication between PAC++ and other systems or languages. The global organization (arithmetics, matrices, polynomials) is presented and in particular we focus on choices that have been made for the conception of *communication interface* and memory management. Finally, we provide experimental performances obtained for some linear algebra functions.

The second part deals with the problem of

parallel computation with algebraic numbers. Recently, the problem of computing the Jordan normal form of a matrix (with entries in a field F, with no restriction) [28] and the Smith normal form of a matrix (with entries in $F[X]$) have been proven to be in NC. The associated algorithms strongly relate to parallel computation in algebraic extensions [28]. We then present how algebraic numbers are integrated in PAC++. This leads to some practical parallel efficient parallelizations of the previous theoretical algorithms. The parallel mechanism we propose is illustrated by the computation of the Jordan normal form.

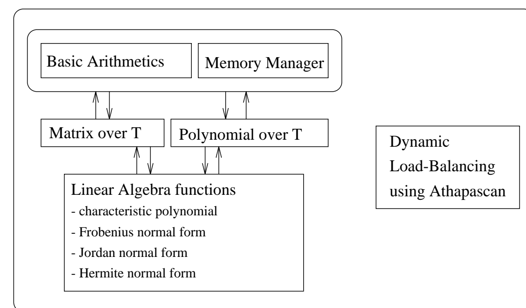

Figure 2: Global organisation of algebraic computing part

2 Designing a parallel library for algebraic computations

Although our library may be used with any parallel environnement, PAC++ also provides communication facilities which can be called from different systems or languages like a unix-style library (see figure 1). It is designed in C++ in order to use some of its useful characteristics such as overloading and inheritance. Furthermore, PAC++ objects are strongly typed.

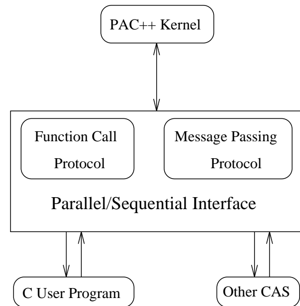

Figure 1: Parallel-sequential interface: the most general method for communicating with PAC++

2.1 Dynamic-load balancing

Algebraic computations are intrinsically dynamic: in most cases, the effective cost of a computation cannot be statically predicted. The main reason for this is that the growth of intermediate coefficients is difficult to estimate precisely before the computation since it relies on many parameters. Therefore, a static parallelization of a general algebraic algorithm (even a highly parallel one) is a complex issue: given a number of processors, a good parallelization consists in evenly distributing the work load, so as to guarantee the best possible processors utilization.

A dynamic load-sharing mechanism is thus necessary in order to decide how the parallelism of a given algorithm can be exploited (i.e., at which level of granularity it is has to work, which computations have to be distributed ...) in order to decrease the total computational time on a given machine. In computer algebra, the cost of an algorithm, evaluated at a given level of granularity, constitutes a significant information that can be dynamically predicted just before the execution of the algorithm. For instance, the cost of a matrix product relates to the dimension of the matrices at the matrix operations level of granularity. This cost prediction is a relevant information which has to be taken into account by the load-balancer in order to achieve an efficient parallelization [27]. Notice that the type of cost information is used in some other parallel systems for computer algebra. In PAC++, this information is handled at a relatively fine grain level, since target computational supports are fine grain parallel machine. In DSC [6], this information is also used to determine the mapping of the parallel processes, but at a coarser grain.

2.2 Parallel programming model

Although sequential algorithms can be used in parallel programming models based on message passing with functions for packing and unpacking structures, all parallel algorithms of PAC++ are designed for using asynchronous *Remote Procedure Call* to be dynamically load-balanced with runtime-support [26].

A parallel program in PAC++ is a set of processes, called *servers*, with the ability to do algebraic computation. This set of servers are mapped on physical processors. Each of these servers has several entry-points that are called *services*. An RPC call consists of choosing the processor on which the specific service is to be executed and

the sending of the arguments. The result will be returned to the caller in a packed form.

The runtime support used is called *Athapascan* [22]. Athapascan is a parallel extension of C which includes a dynamic load-balancing mechanism and an adaptative granularity scheme. Generally, parallelizing a computation consists of splitting it into parallel subcomputations, using different rules. Splitting may be halted at each step, reducing to a sequential computation: this defines the granularity of the algorithm. In order to ensure portability and efficiency on parallel architectures, a parallel (work load efficient) algorithm written in Athapascan has to provide a maximal parallelism via a recursive splitting mechanism. At each step of the recursion, an alternative between a finer parallel algorithm (expressed by asynchronous RPC) and a sequential one is proposed. The threshold for the decision of a parallel algorithm depends not only of the dimension, but also on the characteristics of the parallel machine and of its work load, which are known by the load balancing mechanism.

An example of our "parallel programming paradigm" is the following: (**Algorithm** denotes a *service*):

```
Algorithm (input, output)
{
 if (threshold(input) = ws_local)
   // --- Sequential computation
   Sequential_Algorithm(input, output);
 else {
   // --- Parallel computation
   // Step 1 : splitting
   Split (input, input1, input2);

   // Step 2 : parallel computation with
   // recurrent calling
   S1 = Spawn( Algorithm, input1, result1);
   S2 = Spawn( Algorithm, input2, result2);

   // Step 3 : fusion
   Wait (S1, S2) ;
   output = Fusion (result1, result2) ;
 }
}
```

2.3 Splitting structures

For functional oriented algorithms the most important operators are the *split* operator which splits any PAC++ structure in order to process it concurrently, and the *merge* operator its counterpart. In this section, we focus on the *split* operator.

All PAC++ structures have their own version of the function *split*. The main argument of the function *split* is the number of sub-structures that must be returned and an array of percentages which precises the part of the total structure in each sub-structure. In order to minimize memory copies the split function performs logical copies. From an implementation point of view, this is realized by *in place* classes which are basic tools for manipulating *in place* effective structures.

For example, given one vector **Vect** over the rationals, the following example shows how to split and communicate two PAC++ sub-vectors to appropriate *services* by asynchronous remote procedure calls:

```
SubVector<Rational> List[2] ;
Buffer B1,B2 ;
Buffer Res1,Res2 ;
Synchro syn_pt1, syn_pt2 ;

// Step 1: Splitting
// Split 'List' in two parts (20% and 80%)
Vect.split(List, 2, 0.2, 0.8) ;

// The two sub-vectors are packed:
B1 << List[0] ; B2 << List[1] ;

// Step 2: Parallel computation
// asynchronous spawn of two services
// on servers 'serv1' and 'serv2'
syn_pt1= Spawn(serv1, service1, B1, Res1);
syn_pt2= Spawn(serv2, service2, B2, Res2);

// Step 3: Fusion
// the results are unpacked in place in
// vector 'Vect'
WaitSpawn ( synchropt1 ) ;
WaitSpawn ( synchropt2 ) ;
Res1 >> List[0] ; Res2 >> List[1] ;
```

2.4 Pack and Unpack functions

Any PAC++ structure which has to be communicated (whatever the model used, e.g. point to point, RPC ...) is packed by its sender in a communication buffer. This buffer is transmitted. The structure is restored by unpacking the buffer, as shown in the above example.

The pack/unpack mechanism is implemented in a recursive manner. Packing on a high level structure will recursively call pack functions on the sub-level structures. For instance, the packing of a matrix begins with row and column dimensions and finishes by a recursive packing of the coefficients using a C-style array storage by rows.

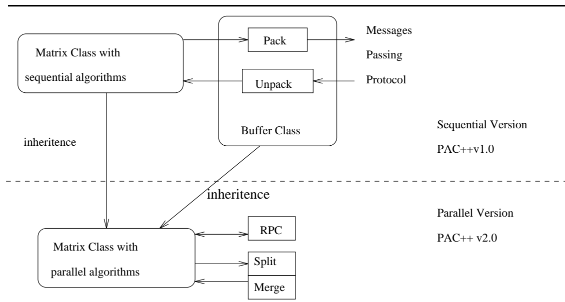

Figure 3: Matrix example to PAC++ object organization of parallel/sequential algorithms

2.5 Memory Management

In order to hold data structures, PAC++ uses a *reference counting* with *free-lists* strategy for memory management.

Such a mechanism is usually not used, since it could be difficult to assert the following invariant "*the reference counts are all correct*". But in C++, three operations are required on objects: initialization, assignment, destruction. In PAC++, any object have a specific definition of this three operators. The developer writes one version of this operators for each class which increase or decrease reference counters and free memory. Then, the C++ compiler automatically generates a call to a constructor for creating an object and to a destructor when the flow control leaves a C++ instruction block.

This three critical operators of C++ are implemented in order to minimize memory copies. The recopy constructor does a logical copy; the destructor does a logical free and the assignment operator implements the following specification: if an object `A` is assigned to `B`, such as `A = B`, all further modifications on `A` or `B` do not modify the other. For classes that can not have member functions which modify internally the data, the assignment operator is implemented by using a logical copy. This is the case for all the basic arithmetics. For other classes, the assignment operator is implemented in a recurrent manner by assignment each of element.

For example, and since no operator can modify the internal representation of an arbitrary precision integer, the assignment between two integers is implemented as a logical copy. Otherwise, for an object matrix over integer, the access element operator

```
Integer& operator() (const int, const int)
```

can be used in read and write mode, so the matrix assignment operator is implemented by creation of a new matrix of the same size in which each coefficient is affected.

The most frequently required bloc sizes are treated by the memory manager as special cases in a fast way.

In the parallel version of our library, each server supports a kernel consisting of a version of the memory manager and of a version of all arithmetics (over the integers, rationals, floating points, matrices and polynomials). Presently PAC++ does not support shared objects between processors.

2.6 Passing arguments by value

In PAC++, the semantics for passing arguments to a function is by value. The cost of such a passing can become very important for large sized objects. This is why in PAC++, all objects are implemented as pointers to actual representations. This customized encapsulation is commonly used in C++ programming and achievies high performance to our library. In order to satisfy the semantics of passing by value, the C++ type specificator `const` must be used.

For example, consider a `sqr` function which takes a matrix in input and returns its square. The canonical form for this kind of declarations is:

```
Matrix sqr (const Matrix& A)
```

We use the reference specifier `&` in order to optimize the code so that the compiler does not generate calls to the recopy constructor.

2.7 Logical organization

The arithmetic part of the PAC++ library is divided in three levels: the first consists of the basic arithmetics on arbitrary precision integers, rationals, floating point numbers and residue number systems.

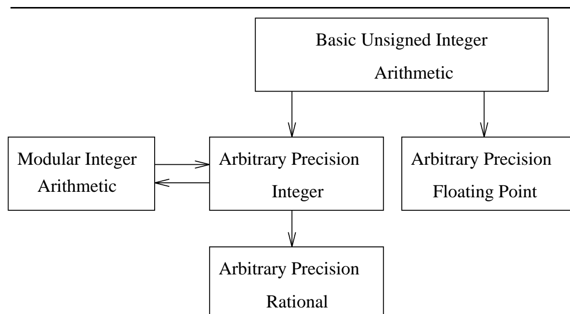

Figure 4: PAC++ Basic arithmetics, hierarchical dependence

As shown figure 4, the arbitrary precision arithmetics over integers and floating point numbers are based on unsigned arbitrary precision integer. The performance of this module is very critical for the overall performance of the library. This module is currently implemented on the GNU Multi-precision functions on non-negative integers (see Gmp in [14]).

The second level concerns all structures which are the representation of a T-algebra (such as matrices or polynomials), based on an algebraic structure represented by T. In the first version of PAC++, this module is made of C++ template definitions of classes.

The third level contains all non-basic functions using the two previous levels. One can find here the linear algebra module.

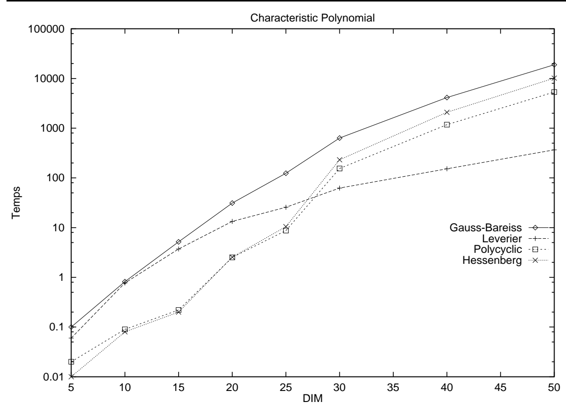

Figure 5: With sparse matrix input (10 percent of non-zero coefficients per row)

2.8 Experimental results

In this section, we give some performance results of implemented algorithms. The theoretical results are given in $O^{\sim}$ notation (the so-called O-smooth), which means that we hide logarithmic factors.

Since the PAC++ arithmetics are based on Gmp, the sequential time complexity for multiplication is the one of Karatsuba algorithm [16], $O(n^{1.59})$ bit operations. In fact, the multiplication algorithm of Gmp automatically switches between the standard algorithm in $O(n^2)$ and the Karatsuba one, depending on the sizes of the input numbers (about ten machine-words). In the following, we will use $O(n^{1.59})$ to give the cost of more elaborated PAC++ algorithms.

In the implementation of the polynomial arithmetic, the multiplication between two polynomials of degree n is done in sequential time $O(n^2)$.

2.8.1 Characteristic polynomial

The PAC++ library implements the $O(n^{2.81})$ arithmetic operations Strassen matrix multiplication with automatic threshold for switching between standard and the Strassen algorithms. In PAC++, four methods for computing characteristic polynomial of square matrices can be found. Two methods are a Gauss-Bareiss [1], computation of the determinant of the characteristic matrix, the Leverier-Souriau-Faddeeva algorithm [10, 11, 4]. The two other methods use intermediate similar matrix forms (Hessenberg [23, 3], and polycyclic [21, 17, 21] form) and deduce the characteristic polynomial by recurrent relations.

Note that the Leverier-Souriau-Faddeeva method with standard arithmetic of PAC++ is the best one (on any type of matrix for sufficiently large dimensions). This can be easily deduced from the cost of matrix-vector multiplications.

In contrast, with a modular integer arithmetic, the best methods are polycyclic or Hessenberg methods. In fact, in this methods only the sizes of the intermediate coefficients are $O(n^2)$ bits. Since the coefficients in the characteristic polynomial are $O(n)$ bits length, only $O(n)$ primes are needed to recover the values from residues.

2.8.2 Smith normal form of polynomial matrix

We now provide some practical results concerning the computation of the Smith normal form of polynomial matrices. Since it is not currently our purpose to study the Smith normal form, we only briefly recall some elementary properties. The form is generally defined over a principal domain. For an input matrix A of dimension n, the Smith form is a diagonalization of A which is entirely computed within the domain of the entries, *i.e.* using unimodular transformations [20].

Computing the Smith normal form may require a huge amount of time. For a random input 8×8 matrix of degree 4 with absolute values of integer coefficients less than 3, the default Maple routine will run more than ten hours CPU on a Sun4 !

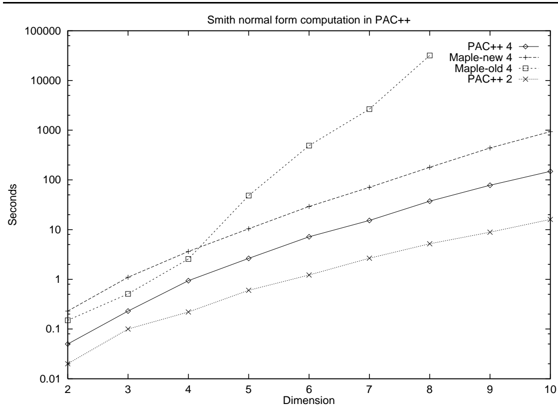

Figure 6: Smith normal form computations over $Q[x]$.

We compare below two different algorithms over $Q[x]$. Let us notice that since our input matrices will be chosen at random, the cost for computing the Smith normal form will roughly be the cost for computing the Hermite normal form. The first algorithm, say STD-algorithm, is the default in Maple. It consists in a standard elimination process over the polynomials [15]. The second algorithm, say SYLV-algorithm, is the one proposed in [30]. It is based on generalized subresultants. The form is computed by diagonalizing over Q a "generalized Sylvester" matrix constructed from the input polynomial matrix.

Figure 6 gives average computational times for random input matrices of degree 2 or 4 whose entries are integers absolutely bounded by 3. Clearly, the STD-algorithm (Maple-old) is prohibitive even for small dimensions. For an 8×8 matrix of degree 4 it runs nearly 1000 times slower than SYLV-algorithm (Maple-new). From this figure, we also conclude to the superiority of PAC++ over Maple. Still for an 8×8 matrix, the Maple code runs in 940 seconds while the same code in PAC++ runs in only 150 seconds.

3 Algebraic number

Many problems in computer algebra involve computations with roots of polynomials, i.e. arithmetic in the algebraic closure of fields. In the following, F is an arbitrary commutative field, $P(\alpha)$ is an univariate polynomial of degree n over F with $l \leq n$ distinct roots $\alpha_1, \ldots, \alpha_l$. A first way to compute with the roots of P is to assume that P is irreducible over F. In practice, the factorization of P can be difficult and expensive. In [5], an alternative way, called D5, is presented which allows computation in $\tilde{F}$, the algebraic closure of F, with no need of factorization. Even if only sequential versions of D5 are available [7, 13], its working is intrinsically parallel, and the same model has been used to show the intrinsic parallelism of some linear algebra problems which relate to eigenvalues [28]. The aim of this part is to explain how parallel computation with algebraic numbers is handled in PAC++. First, we recall how algebraic numbers are represented in D5 [7], and how parallelism appears while computing with such numbers. Next, the related primitives are presented and the implementation is detailed. We conclude by an example, where algebraic numbers appear naturally: the parallel computation of the Jordan normal form.

3.1 Computation with algebraic numbers

Given F, computing in $\tilde{F}$ reduces to the construction of a tower of subfields F_i $(1 \leq i \leq n)$

$$F \subset F_1 \subset \ldots \subset F_n \subset \tilde{F}$$

such that each F_i is a simple algebraic extension of F_{i-1} by a root α_i of a univariate polynomial P_i with coefficients in F_{i-1} [8]. F_i is denoted $F_{i-1}(\alpha_i)$. Usually, the elements of F_i are represented as polynomials in $(F_{i-1}[\alpha]/(P(\alpha))$. This is the common way algebraic extensions appear in computer algebra. The problem is that generally P_i is reducible and characterizes not only one root α_i, but a set of roots. Therefore, the result of a computation may not be the same for all the roots of P_i, typically for boolean output of `=?0` (*i.e.* test to zero) gates for an arithmetic circuit over $\tilde{F}$ [12]. As an example, let α denote a root of $P = x^5 - 2x^3 - x^2 + 2$ in $\tilde{Q}$, the algebraic closure of the field of the rationals. The answer to the question $\alpha^2 - 2 =?0$ may be *true* or *false*, depending on the chosen root α of P. This leads to an automatic discussion, called *splitting* in D5, which reduces to a partial factorization of P. Notice that this partial factorization does not require polynomial factorization, but only polynomial gcd computations, and thus is in NC^2 [28]. In the sequential implementation of D5, the test $\alpha^2 - 2 =?0$ in the previous example would return: "Either α is a root of $x^2 - 2 = 0$ and the answer is *true*, or α is a root of $x^3 - 1$ and the answer is *false*."

3.2 Handling algebraic numbers

The solution proposed in [7] to compute in $\tilde{F}$ is the introduction of a class, denoted by `Alg<F>` in

PAC++. Since basic field operations (`+,-,*,/`) in `Alg<F>` are performed exactly like in D5 using polynomial remainder and gcd computations, we focus only on the splitting mechanism (which corresponds to the `=?0` test operation) and its parallelization. In D5, the user can define a symbol to be a generic root of a polynomial. To manage splittings that occur in a computation, a discussion package is adjunctive to the top of the class `Alg<F>`, with one function **AllCases** that allows the user to obtain the list of results, one result per splitting. From a parallel point of view, two kinds of computations with algebraic numbers may be distinguished:

1. situations involving only one root of P, whatever this root is. For instance, implementation of Householder algorithm to compute the QR factorization of a matrix involves one of the root of a polynomial of degree 2 at each step. Another example is a *sequence* of computations performed for every root of a polynomial but one root after the other. In such situations, at the end of an algorithm, the result has to be valid for at least one of the roots of P, and a new polynomial characterizing this root is expected. For instance, the program: " Compute $\alpha^2 - 2$ =?0 for α one of the roots of $x^5 - 2x^3 - x^2 + 2$" could deliver as output: *false* if $\alpha^3 - 1 = 0$.
2. situations involving a parallel computation for every root of a polynomial P. The expected output of such a computation is then a set of results, each result being associated with a polynomial that characterizes the roots of P that leads to this result. For example, the program: "For each α root of $x^5 - 2x^3 - x^2 + 2$, compute $\alpha^2 - 2$ = ?0" should deliver in a D5 manner: *true* if $\alpha^2 - 2 = 0$ and *false* if $\alpha^3 - 1 = 0$.

To deal with the first case, a constructor is provided by PAC++:

```
Alg<K> OneRoot (Polynom< Alg<K> > P)
```

It allows the construction of an algebraic number that represents **one** root (always the same) of the polynomial argument P with possibly algebraic number coefficients. When a splitting occurs on such an algebraic number, a random choice is performed by the system to ensure the validity of the result. The selector:

```
Polynom< Alg<K> > GetReductionRule ()
```

returns the polynomial, factor of the initial P, that currently characterizes the algebraic number. Since parallel computations are possible (and recommended !) in PAC++, two concurrent processes, dealing with an algebraic number α may induce different choices in the splittings. To ensure coherency, the splitting is then performed in global mutual exclusion. Notice that those two primitives allow the user to manage "by hand" sequential computations in a D5 manner.

The second scheme of computation is intrinsically parallel: the same program has to be executed for all the roots. For such a scheme, a parallel iterator is provided:

```
ParForAllRoots(a, Polynom< Alg<K> > P,
   fn, in args, int& n, out& res[] )
```

where `fn` is a service having the following the prototype:

```
service fn (in args, out res)
```

Such a loop provides the execution of the *service* `fn` for the groups of roots `a` of `P` that behave differently (branching may occur). When this loop terminates, the number of different splittings is returned by n, together with the array of the results returned by the different executions of the service `fn`.

When a splitting occurs, a new execution of the service is spawned, while the current execution keeps on: both executions compute with a different subset of roots of P. Due to parallel computation, two concurrent processes may be created in the body of `fn`. Since those processes have to work with the same subset a of roots, splitting of this subset into two new subsets is made, as previously in mutual exclusion.

3.3 A case study: Jordan normal form of matrices

Let A be a square matrix of dimension n over a field F. We briefly recall the scheme of the NC algorithm proposed in [28] to compute the Jordan normal form of A. Let $\xi_A(x)$ be the multiplicity free (or square free if the field is perfect) greatest factor of the characteristic polynomial of A. The algorithm computes $\Delta_i^{(k)}$, the greatest factor of ξ_A such that $\text{rank}((A - \lambda Id)^k) = i$ for $0 \leq i, k \leq n$. Then, a generalized multiplicity free basis is computed for the set of polynomials $(\Delta_i^{(k)})_{i,k}$. Let D_j be a polynomial in the resulting set. For any k, there exists an unique integer $i_{j,k}$ such that D_j divides $\Delta_{i_{j,k}}^{(k)}$. This means that $\text{rank}((A - \lambda Id)^k) = i_{j,k}$ for each of the roots of D_j. Now, the number of Jordan blocks of dimension k associated to each of the roots of D_j is $(i_{j,k+1} - 2i_{j,k} + i_{j,k-1})$. This algorithm works in

time $O(\log^3 n)$, its work load being $O(n^5 M(n))$ where $M(n)$ denotes the number of processors needed to compute in logarithmic time the product of two square matrices of dimension n with entries in F.

The main work load comes from the computation of the $\Delta_i^{(k)}$. In [28], those polynomials are directly computed using a slight modification of Mulmuley's rank algorithm [19]. However, even if this leads to a polylog time parallel algorithm, on the one hand it is far not workload efficient, and on the other hand it makes that the computation of the Jordan form strongly relates to Mulmuley's algorithm. Using the previous primitives for handling algebraic numbers leads to a more efficient algorithm, directly built on a *black box* to compute the rank in any field, whatever the implemented rank algorithm is. Since field operations are provided in `Alg<F>` (`+, -, *, /, =?0`), this rank algorithm may be used to compute the rank of a matrix with entries in `Alg<F>`. During the computation, splittings may occur that can be handled thanks to "ParForAllRoots" iterator. At the end of the body of this iteration, "GetReductionRule" returns the algebraic relation that characterizes roots that lead to the current value for the rank.

Consider now the implementation of the algorithm. Let m be the highest multiplicity in the minimal polynomial P_A of A. $(A - \lambda Id)^k$ is of interest only if $k \leq m$. Those matrices may be directly computed from the computation of the powers A^k $(1 \leq k \leq m)$ (prefix scheme). The call "mat(λ,k)" is assumed to compute the matrix: $\sum_{i=0}^{k} C_k^i \lambda^i A^{k-i}$ using the previously computed powers of A. Assuming that polynomials $\Delta[n][n]$ are initialized to the constant 1 polynomial, the algorithm may be then written in pseudo PAC++:

ParForAllRoots(λ, P_A) do
 ParFor $k = 1, m$ do
  r = rank(mat(λ, k)) ;
  $\Delta[r][k] = \Delta[r][k] *$ GetReductionRule(λ)
;
 EndParFor
EndParForRoots

Since different processes may read and modify the same global variable $\Delta[r][k]$, modifications to this variable have to be performed in mutual exclusion.

The main advantage of this algorithm, related to the theoretical one, is that its permits to use an efficient parallel algorithm to compute the rank. The use of the algebraic number arithmetic and of the lazy mechanism of splitting generates exactly the parallelism needed for the application. Each arithmetic operations in F in the theoretical algorithm has been replaced by an arithmetic operation in `Alg<F>`. This means that, since the degree of P_A is bounded by n, the cost of each arithmetic operation is $O(GCD(n))$ operations in F (here, $GCD(n)$ denotes the workload of the parallel algorithm to compute the gcd of two polynomials of degree n in $F[x]$). If $R(n)$ is the workload need to compute rank of a square matrix of order n, then the whole workload is $O(mnGCD(n)R(n))$.
From a practical point of view, using algebraic numbers in this example leads to a coarse grain parallelism, that may be efficiently used on a parallel machine.

4 Conclusions

Currently parallel computers are used to solve high dimension problems of algebraic computing. In PAC++, the sequential/parallel algorithm approach, with dynamic load-balancing, provides an interested concept to develop efficient and scalable libraries for massively parallel computers. More and more numerical methods are going to use algebraic computation in order to solve ill-conditioned problems, this is why the general interface of PAC++ will be developed to allow easy communications between Lapack (Lapack++) and PAC++.

Currently, we are going to develop some parallel linear algebra functions and algebraic number arithmetic on our 32-processor parallel computer IBM SP1. The experiments with algebraic numbers should agree with theoretical results and the proposed tool is expected to be powerful.

References

[1] E.H. Bareiss. Computational solution of matrix problems over an integral domain. *J. Inst. Math. Appl.*, 10:68–104, 1972.

[2] B.W. Char. Progress report on a system for general purpose parallel symbolic algebraic computations. In *ISSAC'90, Tokyo, Japan.* ACM Press, pp 96–103, 1990.

[3] H. Cohen. *A course in Computational Algebraic Number Theory.* Springer-Verlag, 1993.

[4] L. Csanky. Fast parallel matrix inversion algorithms. *SIAM J. Comput.*, 5:618–623, December 1976.

[5] J. Della Dora, C. Dicrescenzo, and D. Duval. About a new method for computing in algebraic number fields. In *Proc. EUROCAL'85*, LNCS 204, Springer Verlag, pages 289–290, 1985.

[6] A. Diaz, E. Kaltofen, K. Schmitz, and T. Valente. DSC A System for Distributed Symbolic Computation. In M. Watt, editor, *ISSAC'91*, pages 324–333. ACM Press, 1991.

[7] C. Dicrescenzo and D. Duval. Algebraic extensions and algebraic closure in Scratchpad II. In *Proc. ISSAC'88*, LNCS 358, Springer Verlag, pages 440–446, 1988.

[8] D. Duval. *Diverses questions relatives au calcul formel avec des nombres algébriques.* Thèse de Doctorat d'Etat, Université de Grenoble, France, 1987.

[9] H. Hong et al. *PACLIB User Manual.* Research Institute for Symbolic computation (RISCC-Linz), Johannes Kepler University, Linz, Austria, October 1992.

[10] V.N. Faddeev. *Computational Methods of Linear Algebra.* Dover Publications Inc, New York, 1959.

[11] F.R. Gantmacher. *Théorie des Matrices.* Jacques Gabay, 1957.

[12] J. von zur Gathen. Algebraic complexity theory. *Ann. Rev. Comput. Sci.*, 3:317–347, 1988.

[13] T. Gómez-Díaz. *Quelques applications de l'évaluation dynamique* . PhD thesis, Université de Limoges, France, 1994.

[14] T. Granlund. *The GNU Multiple Precision Arithmetic Library.*

[15] R. Kannan. Solving systems of linear equations over polynomials. *Theoretical Computer Science*, 39:69–88, 1985.

[16] A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. Technical Report AFCRL-65-758, Air Force Cambridge Research Laboratory, 1962.

[17] W. Keller-Gehrig. Fast Algorithms for the Characteristic Polynomial. *Theoretical Computer Science*, 36:309, 1985.

[18] W. Küchlin. A parallel sac-2 based on threads. Technical report, Computer and Information Science, Ohio State University, Columbus, April 1990.

[19] K. Mulmuley. A fast parallel algorithm to compute the rank of a matrix over an arbitrary field. *Combinatorica*, 7(1):101–104, 1987.

[20] M. Newman. *Integral Matrices.* Academic Press, 1972.

[21] P. Ozello. *Calcul exact des formes de Jordan et de Frobenius d'une matrice.* PhD thesis, Université Scientifique Technologique et Médicale de Grenoble, 1987.

[22] B. Plateau and al. Apache presentation - athapascan. Technical Report 93-1, IMAG Institute, Grenoble France, October 1993.

[23] T.M. Rao. Error-free computation of characteristic polynomial of a matrix. *Comp. and Math. with Appl.*, (4):61–65, 1978.

[24] J.L. Roch. The PAC System : General Presentation. In *Symposium CAP 2, Univ. Cornell - N-Y Springer-Verlag, LNCS 584*, 1990.

[25] J.L. Roch, F. Siebert, P. Sénéchaud, and G. Villard. Computer Algebra on a MIMD machine. *ISSAC'88, LNCS 358 and in SIGSAM Bulletin, ACM*, 23/11, p.16-32, 1989.

[26] J.L. Roch, A. Vermeerbergen, and G. Villard. Cost prediction for load-balancing: application to algebraic computations. In *CONPAR 92, Lyon France*, volume 634 of *LNCS*, pages 467–478, September 1992.

[27] J.L. Roch, A. Vermeerbergen, and G. Villard. A new load-prediction scheme based on algorithmic cost functions. In *CONPAR 94, Linz Austria*, LNCS, September 1994.

[28] J.L. Roch and G. Villard. Parallel computations with algebraic numbers, a case study: Jordan normal form of matrices. In *PALE'94, Athens Greece*, LNCS, July 1994.

[29] K. Siegl. ——Maple—— - A system for parallel symbolic computations. In *Parallel Systems Fair at the 7th International Parallel Processing Symposium, Newport Beach, CA*, April 1993.

[30] G. Villard. Computation of the Smith normal form of polynomial matrices. In *ISSAC'93, Kiev, Ukraine.* ACM Press, pp 209–217, July 1993.

[31] S.M. Watt. *Bounded Parallelism in Computer Algebra.* PhD thesis, Univ. Waterloo, Ontario, 1985.