Generic design of Chinese remaindering schemes

Jean-Guillaume Dumas*
Université de Grenoble
Laboratoire Jean Kuntzmann,
umr CNRS 5224,
BP 53X, F38041 Grenoble, France.
Jean-Guillaume.Dumas@imag.fr

Thierry Gautier†
Université de Grenoble
Laboratoire LIG
51, av. Jean Kuntzmann,
F38330 Montbonnot, France.
Thierry.Gautier@inrialpes.fr, JLRoch@imag.fr

Jean-Louis Roch
Université de Grenoble
Laboratoire LIG
51, av. Jean Kuntzmann,
F38330 Montbonnot, France.
JLRoch@imag.fr

ABSTRACT
We propose a generic design for Chinese remainder algorithms. A Chinese remainder computation consists in reconstructing an integer value from its residues modulo non coprime integers. We also propose an efficient linear data structure, a radix ladder, for the intermediate storage and computations. Our design is structured into three main modules: a black box residue computation in charge of computing each residue; a Chinese remaindering controller in charge of launching the computation and of the termination decision; an integer builder in charge of the reconstruction computation. We then show that this design enables many different forms of Chinese remaindering (e.g. deterministic, early terminated, distributed, etc.), easy comparisons between these forms and e.g. user-transparent parallelism at different parallel grains.

Categories and Subject Descriptors
D.2.2 [Software Engineering]: Design Tools and Techniques—Object-oriented design methods; I.1.2 [Computing Methodologies]: Symbolic and Algebraic Manipulation—Algorithms; G.4 [Mathematics of Computing]: Mathematical Software—Algorithm design and analysis

1. INTRODUCTION
Modular methods are largely used in computer algebra to reduce the cost of coefficient growth of the integer, rational or polynomial coefficients. Then Chinese remaindering (or interpolation) can be used to recover the large coefficient from their modular evaluations by reconstructing an integer value from its residues modulo non coprime integers.

*Part of this work was done while the first author was visiting the Claude Shannon Institute and the University College Dublin, Ireland, under a CNRS grant.
†Part of this work was done while the second author was visiting the ArTeCS group of the University Complutense, Madrid, Spain.

LINBox [9] is an exact linear algebra library providing some of the most efficient methods for linear systems over arbitrary precision integers. For instance, to compute the determinant of a large dense matrix over the integers one can use linear algebra over word size finite fields [10] and then use a combination of system solving and Chinese remaindering to lift the result [13]. The Frobenius normal form of a matrix is used to test two matrices for similarity. Although the Frobenius normal form contains more in formation on the matrix than the characteristic polynomial, most efficient algorithms to compute it are based on computations of characteristic polynomial (see for example [23]). Now the Smith normal form of an integer matrix is useful e.g. in the computation of homology groups and its computation can be done via the integer minimal polynomial [12]. In both cases, the polynomials are computed first modulo several prime numbers and then only reconstructed via Chinese remaindering using precise bounds on the integer coefficients of the integer characteristic or minimal polynomials [18, 8].

An alternative to the deterministic remaindering is to terminate the reconstruction early when the actual integer result is smaller than the estimated bound [14, 12, 20]. Thereafter the reconstruction stabilizes for some modular iterations, the computation is stopped and gives the correct answer with high probability.

In this paper we propose first in section 2 a linear space data structure enabling fast computation of Chinese reconstruction, alternative to subproduct trees. Then we propose in section 3 to structure the design of a generic pattern of Chinese remaindering into three main modules: a black box residue computation in charge of computing each residue; a Chinese remaindering controller in charge of launching the computation and of the termination decision; an integer builder in charge of the reconstruction computation. We show in section 4 that this design enables many different forms of Chinese remaindering (e.g. deterministic, early terminated, distributed, etc.) and easy comparisons between these forms. We show then in section 5 that this structure provides also an easy and efficient way to provide user-transparent parallelism at different parallel grains. Any parallel paradigm can be implemented provided that it fulfills the defined controller interface. We here chose to use KAAPI [16] to show the efficiency of our approach on distributed/shared architectures.

1http://linalg.org
2http://kaapi.gforge.inria.fr
2. RADIX LADDER: LINEAR STRUCTURE FOR FAST CHINESE REMAINDERING

2.1 Generic reconstruction

We are given a black box function which computes the evaluation of an integer \( R \) modulo any number \( m \) (often a prime number).

To reconstruct \( R \), we must have enough evaluations \( r_j \equiv R \mod m_j \) modulo coprimes \( m_j \). To perform this reconstruction, we need two by two liftings with \( U \equiv R \mod M \) and \( V \equiv R \mod N \) as follows:

\[
R_{MN} = U + (V - U) \times (M^{-1} \mod N) \times M.
\]

We will need this combination most frequently in two different settings: when \( M \) and \( N \) have the same size, and when \( N \) is of size 1. The first generic aspect of our development is that for both cases, the same implementation can be fast.

We first need a complexity model. We do not give much details on fast integer arithmetic in this paper, instead our point is to show the genericity of our approach and that it facilitates experiments in order to obtain good practical efficiency with any underlying arithmetic. Therefore we propose to use a very simplified model of complexity where cost of the division is also bounded by \( O(\alpha) \) and there exists \( m_n \) such that the complexity of integer multiplication of size \( l \) can be bounded by \( m_n l^{\alpha} \) and there exists \( \alpha \) such that the complexity of integer multiplication of size \( l \) is of size \( 1 \). The first generic aspect of our development is of that fact in a parallel setting in section 5.

We denote by \( m_n l^{\alpha} \) the complexity of \( \gcd \), 

\[
\gcd(a, b) = b \gcd(a, b \mod a)
\]

With this in mind we compute formula (1) with one multiplication modulo as follows:

\[
R_{MN} = U + (V - U) \times (M^{-1} \mod N) \times M.
\]

Now, if the formula (1) is computed via algorithm 1 and the operation counts uses column “Mul.” for multiplication and “Div./Gcd.” for division/inverse/modulo/gcd, then we have the complexities given in column “CRT” of table 1.

<table>
<thead>
<tr>
<th>Size of operands</th>
<th>Mul.</th>
<th>Div./Gcd.</th>
<th>CRT</th>
</tr>
</thead>
<tbody>
<tr>
<td>( l \times 1 )</td>
<td>( \frac{3l}{m_n} )</td>
<td>( 2(\alpha + d_n)l^{\alpha} + O(l) )</td>
<td>( \frac{9l}{\alpha} + O(1) )</td>
</tr>
<tr>
<td>( l \times l )</td>
<td>( \frac{m_n l^{\alpha}}{\alpha} )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Integer arithmetic complexity model

2.2 Radix ladder

Fast algorithms for Chinese remaindering rely on reconstructing pairs of residues of the same size. A usual way of implementing this is via a binary tree structure (see e.g. figure 1 left). But Chinese remaindering is usually an iterative procedure and residues are added one after the other. Therefore it is possible to start combining them two by two before the end of the iterations. Furthermore, when a combination has been made it contains all the information of its leaves. Thus it is sufficient to store only the partially recombined parts and cut its descending branches. We propose to use a radix ladder for that task. A radix ladder is a ladder composed of successive shelves. A shelf is either empty or contains a module and an associated residue, denoted respectively \( M_i \) and \( U_i \) at level \( i \). Moreover, at level \( I \), are stored only residues or modulo of size \( 2^i \). New pairs of residues and moduli can be inserted anywhere in the ladder. If the shelf corresponding to its size is empty, then the pair is just stored there, otherwise it is combined with occupant of the shelf, the latter is dismissed and the new combination tries to go one level up as shown on algorithm 2.

\[
\begin{align*}
\text{Algorithm 2 RadixLadder.insert}(U, M) \\
\text{Input: } U \equiv R \mod M \text{ and a Radix ladder} \\
\text{Output: } \text{Insertion of } U \text{ and } M \text{ in the ladder.} \\
1: \text{for } i = \text{size}(M) \text{ while Shelf}[i] \text{ is not empty do} \\
2: \text{ } \text{ } U, M := \text{RECONSTRUCT}(U \mod M, U_i \mod M) \\
3: \text{ } \text{ } \text{Pop Shelf}[i] \\
4: \text{ } \text{ } \text{Increment } i \\
5: \text{ } \text{ } \text{end for} \\
6: \text{Push } U, M \text{ in Shelf}[i] \\
\end{align*}
\]

Then if the new level is empty the combination is stored there, otherwise it is combined and goes up ... An example of this procedure is given on figure 1.

![Figure 1: A residue going up the radix ladder](http://gmplib.org/gmp-man-4.3.0.pdf)
Algorithm 3 RADIXLadder.merge
Input: Two radix ladders $RL_1$ and $RL_2$.
Output: In place merge of $RL_1$ and $RL_2$.
1: for $i = 0$ to size($RL_2$) do
2: $RL_1$.insert($RL_2$.Shelf[i]);
3: end for
4: Return $RL_1$

3. A CHINESE REMAINDERING DESIGN PATTERN
The generic design we propose here comes from the observation that there are in general two ways of computing a reconstruction: a deterministic way computing all the residues until the product of moduli reaches a bound on the size of the result; or a probabilistic way using early termination. We thus propose an abstraction of the reconstruction process in three layers: a black box function produces residues modulo small moduli, an integer builder produces reconstructions using algorithm 2, and a Chinese remaindering controller commands them both.

Here our point is that the controller is completely generic where the builder may use e.g. the radix ladder data structure proposed in section 2 and has to implement the termination strategy.

3.1 Black box residue computation
In general this consists in mapping the problem from $\mathbb{Z}$ to $\mathbb{Z}/m\mathbb{Z}$ and computing the result modulo $m$. Such black boxes are defined e.g. for the determinant, valence, minpoly, char-Z contains also the whole interface of the Builder.

control is a simple loop. Algorithm 4 shows this loop which other functionalities are provided by any builder. Then the inination and output the reconstructed element. The gener-
ations of moduli and the black box are parameters and the algorithmations that there are in general two ways of computing a re-
construction: a deterministic way computing all the residues until the product of moduli reaches a bound on the size of the result; or a probabilistic way using early termination. We thus propose an abstraction of the reconstruction process in three layers: a black box function produces residues modulo small moduli, an integer builder produces reconstructions using algorithm 2, and a Chinese remaindering controller commands them both.

Here our point is that the controller is completely generic where the builder may use e.g. the radix ladder data structure proposed in section 2 and has to implement the termination strategy.

3.1 Black box residue computation
In general this consists in mapping the problem from $\mathbb{Z}$ to $\mathbb{Z}/m\mathbb{Z}$ and computing the result modulo $m$. Such black boxes are defined e.g. for the determinant, valence, minpoly, char-poly, linear system solve as function objects IntegerModular* (where * is one of the latter functions) in the linbox/solutions directory of LINBOX-1.1.7.

3.2 Chinese remainingder controller
The pattern we propose here is generic with respect to the termination strategy and the integer reconstruction scheme. The controller must be able to initialize the data structure via the builder; generate some coprime moduli; apply the black box function; update the data structure; test for termination and output the reconstructed element. The generations of moduli and the black box are parameters and the other functionalities are provided by any builder. Then the control is a simple loop. Algorithm 4 shows this loop which contains also the whole interface of the Builder.

Algorithm 4 CRA-Control
1: Builder.initialize();
2: while Builder.notTerminated() do
3: $p :=$ Builder.nextCoPrime();
4: $v :=$ BlackBox.apply($p$);
5: Builder.update($v,p$);
6: end while
7: Return Builder.reconstruct();

LINBOX-1.1.7 gives an implementation of such a controller, parametrized by a builder and a black box function as the class ChineseRemainder in linbox/algorithms/cra-domain.h.

The interface of a controller is to be a function class. It contains a constructor with a builder as argument and the functional operator taking as argument a BlackBox, computing e.g. a determinant modulo $m$, and a moduli generator and returning an integer reconstructed from the modular computations. Algorithm 5 shows the specifications of the LINBOX-1.1.7 controller. Then any higher-level algorithm just choose its builder and its controller and pass them the modular BlackBox iteration it wants to lift over the integers.

3.3 Integer builders
The role of the builder is to implement the interface defined by algorithm 4.

There are already three of these implementations in LINBOX-1.1.7: an early terminated for a single residue, an early terminated for a vector of residues and a deterministic for a vector of residues (resp. the files cra-early-single.h, cra-early-multip.h and cra-full-multip.h in the linbox/algorithms directory). Up to now the radix ladder is not a separated class as only this data structure is currently used and as it is simple enough to inherit from one of the latter and modify the behavior of the methods.

Algorithm 5 C++ ChineseRemainder class
1: template<class Builder> struct ChineseRemainder
2: {
3: ChineseRemainder(const Builder& b): bldr(b) {}
4: template<class Function> Integer& operator() (Integer & res,
5: const Function & f,
6: ChineseRemainder & b) {
7: // CRA-Function ...
8: }
9: protected: Builder bldr;
10: }
11: 
12: just choose its builder and its controller and pass them the modular BlackBox iteration it wants to lift over the integers.

3.3 Integer builders
The role of the builder is to implement the interface defined by algorithm 4.

There are already three of these implementations in LINBOX-1.1.7: an early terminated for a single residue, an early terminated for a vector of residues and a deterministic for a vector of residues (resp. the files cra-early-single.h, cra-early-multip.h and cra-full-multip.h in the linbox/algorithms directory). Up to now the radix ladder is not a separated class as only this data structure is currently used and as it is simple enough to inherit from one of the latter and modify the behavior of the methods.

Figure 2: Early termination of a vector of residues via a linear combination

Actually EarlyMultipCRA inherits from both EarlySingleCRA and FullMultipCRA as it uses the radix ladder of FullMultipCRA for its reconstruction and the early termination of EarlySingleCRA to test a linear combination of the residues to be reconstructed as shown on figure 2 The FullMultipCRA has been
implemented so that when a vector/matrix is reconstructed the moduli and some computations are shared among the ladders. We give more implementation details on the early termination strategies in sections 4 and 5.

3.4 Mappers and binders
To further enhance genericity, the mapping of between integer and field operations can also be automatized. If the data structure storing the matrix exposes of binder adaptors generic mappers can be designed. This is the case for the sparse and dense matrices of linbox and a generic converter, using the Givaro/LinBox fields init and convert converters, can be found in linbox/field/hom.h, linbox/algorithm/matrix-hom.h.

Then, to map any function class to the field representation one can use the following generic mapper:

Algorithm 6 C++ Mapper class

```cpp
template<class Data, class Function>
struct Mapper {
    Data m(const Data &d, const Function &h) {}

template<class Field> typename Field::Element operator()(Field f) const {
    typename Data::template rebind<Field>::other Ap;
    Homomorphism::map(Ap, this->A, F);
    return this->g(d, Ap);
}
protected: const Data &A; const Function &g;
};
```

An example of the design usage, here computing a determinant via Chinese remaindering, is then simply:

Algorithm 7 C++ Chinese remaindering scheme

```cpp
// [...] builder, matrix initializations etc.
ChineseRemainder<ModularEarlySingleCRA> ETcra(Builder);
Integer d;
ETcra(d, BBox); // Call to the controller
```

4. TERMINATION STRATEGIES

We sketch here several termination strategies and show that our design enables to modify this strategy and only that while the rest of the implementation is unchanged.

4.1 Deterministic strategy
There Full + CRA.update(v, p) just adds the residues to the ladder; where Full + CRA.notTerminated() tests if the product of primes so far exceeds the precomputed deterministic bound.

4.2 Earliest termination
In a sequential mode, depending on the actual speed of the different routines of table 1 on a specific architecture or if the cost of BlackBox.apply is largely dominant, one can choose to test for termination after each call to the black box. A way to implement the probabilistic test of [12, Lemma 3.1] and to reuse every black box apply is to use random primes as the moduli generator. Indeed then the probabilistic check can be made with the incoming black box residue computed modulo a random prime. The reconstruction algorithm of section 3 is then only slightly modified as shown in algorithm 8 and the termination test becomes simply algorithm 9.

Algorithm 8 EarlySingleCRA.update(v, p)

Global: $U \equiv R \mod M$.
Global: A variable Stabilization initially set to 0.
Input: $v \equiv R \mod p$.
Output: $R_{MN} \equiv R \mod M \times p$.
1: $u \equiv U \mod p$;
2: if $u == v$ then
3: Increment Stabilization;
4: Return $(U, M \times p)$;
5: else
6: $Stabilization = 0$;
7: Return RECONSTRUCT$(U \mod M, v \mod p)$;
8: end if

Algorithm 9 EarlySingleCRA.notTerminated()

1: Return Stabilization < EarlyTerminationThreshold;

In the latter algorithm, EarlyTerminationThreshold is the number of successive stabilizations required to get a probabilistic estimate of failures. It will be denoted ET for the rest of the paper. This is the strategy implemented in LINBOX-1.1.7 in linbox/algorithm/cra-early-single.h. With the estimates of table 1, the cost of the whole reconstruction of algorithm 4 thus becomes

$$\sum_{i=1}^{t} (\text{apply} + 8t + O(1)) = (t + ET)\text{apply} + 4(t + ET)^2 + O(t)$$

where $t = \lceil \log_{32}(R) \rceil$ and $\beta$ is the word size.

This strategy enables the least possible number of calls to BlackBox.apply. It it thus useful when the latter dominates the cost of the reconstruction.

4.3 Balanced termination
Another classic case is when one wants to use fast integer arithmetic for the reconstruction. Then the balanced computations are mandatory and the radix ladder becomes handy. The problem now becomes the early termination. There a simple strategy could be to test for termination only when the number of computed residues is a power of two. In that case the reconstruction is guaranteed to be balanced and fast Chinese remaindering is also guaranteed. Moreover random moduli are not any more necessary for all the residues, only those testing for early termination need be randomly generated. This induces another saving if one fixes the other primes and precomputes all the factors $M_i \times (M_{i-1} \mod M_{i+1})$. There the cost of the reconstruction drops by a factor of 2 from $2(m_o + d_e)\beta^m$ to $(m_o + d_e)\beta^m$.

The drawback is an extension of the number of black box
applications from \( \log_2 \log(R) \) + \( ET \) to the largest power of two immediately superior and thus up to a factor of 2 in the number of black box applies. For the Builder, the update becomes just a push in the ladder as shown on algorithm 10.

Algorithm 10 EarlyBalancedCRA.update(\( v, p \))
1: RadixLadder.insert(\( v, p \));

The termination condition, on the contrary tests only when the number of residues is power of two as shown on algorithm 11.

Algorithm 11 EarlyBalancedCRA.notTerminated()
1: if Only one Shelf, Shelf[i], is full then
2: Set \( U_i \) to Shelf[i] residue;
3: for \( j = 1 \) to EarlyTerminationThreshold do
4: \( p := \text{PrimeGenerator}() \);
5: if (\( U_i \mod p \) ! = \( \text{BlackBox}.apply(p) \)) then
6: Return false;
7: end if
8: end for
9: Return true;
10: else
11: Return false;
12: end if

Then, the whole reconstruction of algorithm 4 now requires:
\[
ET \cdot (\text{apply} + 3 \cdot 2^k) + \sum_{i=0}^{k-1} \frac{2^k}{2^i} (\text{apply} + (m_a + d_a)2^i)\]
\[
+(\text{apply} + 3 \cdot 2^k) = (2^k + k + ET - 1) \cdot \text{apply} + \left(2^k\right)^{\omega} \frac{m_a + d_a}{2^\omega} + O(2^k)
\]

(3)

operations, where now \( k = \lceil \log_2(\log_2(R)) \rceil \).

Despite the augmentation in the number of black box applications, the latter can be useful, in particular when multiple values are to be reconstructed.

Example 1. Consider the Gaussian elimination of an integer matrix where all the matrix entries are larger than \( n \) and bounded in absolute value by \( A_\infty \). Let \( a_\infty = \log_2(A_\infty) \) and suppose one would like to compute the rational coefficients of the triangular decomposition only by Chinese remaindering (there exist better output dependent algorithms, see e.g. [22], but usually with the same worst-case complexity). Now, Hadamard bound gives that the resulting numerators and denominators of the coefficients are bounded by \( \sqrt{2}A_\infty^{\omega} \). Then the complexity of the earliest strategy would be dominated by the reconstruction where the balanced strategy or the hybrid strategy of figure 2 could benefit from fast algorithms:

<table>
<thead>
<tr>
<th>EarlySingleCRA</th>
<th>( O(n^2a_\infty) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>EarlyMultipCRA</td>
<td>( O(n^{\omega+1}a_\infty + n^{2+\omega}a_\infty^2 + n^{2}a_\infty^2) )</td>
</tr>
<tr>
<td>EarlyBalancedCRA</td>
<td>( O(2n^{\omega+1}a_\infty + n^{2+\omega}a_\infty^2) )</td>
</tr>
</tbody>
</table>

Table 2: Early termination strategies complexities for Chinese remaindered Gaußian elimination with rationals

In the case of small matrices with large entries the reconstruction dominates and then a balanced strategy is preferable. Now if both complexities are comparable it might be useful to reduce the factor of 2 overhead in the black box applications. This can be done via amortized techniques, as shown next.

4.4 Amortized termination

A possibility is to use the \( \rho \)-amortized control of [2]: instead of testing for termination at steps \( 2^1, 2^2, \ldots, 2^i, \ldots \) the tests are performed at steps \( \rho(1), \rho(2), \ldots, \rho(\omega), \ldots \) with \( 1 < \rho < 2 \) and \( g \) satisfies \( \forall i, g(i) \leq i \). If the complexity of the modular problem is \( C \) and the number of iterations to get the output is \( b \), [2] give choices for \( \rho \) and \( g \) which enable to get the result with only \( b + \frac{C\rho}{g} \) iterations and extra \( O(f(b)) \) termination tests where \( f(b) = \log_2(b) \).

In example 1 the complexity of the modular problem is \( n^{\omega} \), the size of the output and the number of iterations is \( na_\infty \) so that strategy would reduce the iteration complexity from \( 2n^{\omega+1}a_\infty \) to \( (na_\infty) + o(na_\infty) \) and the overall complexity would then become:

| EarlyAmortizedCRA | \( O(n^{\omega+1}a_\infty + n^{2+\omega}a_\infty^2 + \log(na_\infty)na_\infty) \) |

Indeed, we suppose that the amortized technique is used only on a linear combination, and that the whole matrix is reconstructed with a FullMultipCRA, as in figure 2. Then the linear combination has size \( 2\log(n) + n \cdot a_\infty \) which is still \( O(n \cdot a_\infty) \). Nonetheless, there is an overhead of a factor \( \log(na_\infty) \) in the linear combination reconstruction since there might be up to \( O(\log(na_\infty)) \) values \( \rho(i), \rho(i+1), \ldots \) between any two powers of two. Overall this gives the above estimate. Now one could use other \( g \) functions as long as eq. 4 is satisfied.

\[
\begin{cases}
\rho(i) \sim 2^{\lceil \log_2(\rho(i)) \rceil}, & k(i) = o(\rho(i)) \\
\rho(i+k(i)) - \rho(i) \sim 2^{\lceil \log_2(\rho(i)) \rceil}, & k(i) = o(\rho(i))
\end{cases}
\]

5. Parallelization

All parallel versions of these sequential algorithms have to consider the parallel merge of radix ladders and the parallelization of the loop of the CRA-control algorithm 4. Many parallel libraries can be used, namely OpenMP or Cilk would be good candidates for the parallelization of the embarrassingly parallel FullMultipleCRA. Now in the early termination setting, the main difficulty comes from the distribution of the termination test. Indeed, the latter depends on data computed during the iterations. To handle this issue we propose an adaptive parallel algorithm [5, 24] and use the KAALI library [6, 16]. Its expressiveness in an adaptive setting guided our choice, together with the possibility to work on heterogeneous networks.

5.1 KAALI overview

KAALI is a task based model for parallel computing. It was targeted for distributed and shared memory computers. The scheduling algorithm uses work-stealing [3, 1, 4, 17]; an idle processor tries to steal work to a randomly selected victim processor.
The sequential execution of a KAAPI program consists in pushing and popping tasks to dequeue the current running processor. Tasks should declare the way they access the memory, in order to compute, at runtime, the data flow dependencies and the ready tasks (when all their input values are produced). During a parallel execution, a ready task, in the queue but not executed, may be entirely theft and executed on an other processor (possibly after being communicated through the network). These tasks are called dfg tasks and their schedule by work-stealing is described in [16, 17].

A task being executed by a processor may be only partially theft if it interacts with the scheduler, in order to e.g. decide which part of the work is to be given to the thieves. Such tasks are called adaptive tasks and allows fine grain loop parallelism.

To program an adaptive algorithm with Kaapi, the programmer has to specify some points in the code (using kaapi_stealpoint) or sections of the code (kaapi_stealbegin, kaapi_stealend) where thieves may steal work. To guarantee that parallel computation is completed, the programmer has to wait for the finalization of the parallel execution (using kaapi_steal_finalize). Moreover, in order to better balance the work load, the programmer may also decide to preempt the thieves (send an event via kaapi_preempt_next).

### 5.2 Parallel earliest termination

Algorithm 12 lets thieves steal any sequence of primes.

#### Algorithm 12 ParallelCRA-Control

```plaintext
1: Builder.initialize();
2: while Builder.notTerminated() do
3:     p := Builder.nextCoPrime();
4:     kaapi_stealbegin(splitter, Builder);
5:     v := BlackBox.apply(p);
6:     Builder.update(v, p);
7:     kaapi_finalize_steal();
8:     kaapi_stealend();
9:     if require synchronization step then
10:        while kaapi_nomore_thief() do
11:           (list of v, list of p) := kaapi_preempt_next();
12:           Builder.update(list of v, list of p);
13:        end while
14:     end if
15: end while
16: Return Builder.reconstruct();
```

At line 4, the code allows the scheduler to trigger the processing of steal requests by calling the splitter function. The parameters of kaapi_stealbegin are the splitter function and some arguments to be given to its call. These arguments can e.g. specify the state of the computation to modify (here the builder object plays this role). Then, on the one hand, concurrent modifications of the state of computation by thieves, must be taken care of during the control flow between lines 4 and 8: here the computation of the residue could be evaluated by multiple threads without critical section\(^5\). On the other hand, after line 8, the scheduler guarantees that no concurrent thief can modify the computational state when they steal some work. Remark that both branches of the conditional if at line 9 must be executed without concurrency: the iteration of the list of thieves or the generation of the next random modulus are not reentrant.

The role of the splitter function is to distribute the work among the thieves. In algorithm 13, each thief receives a coPrimeGenerator object and the entrypoint to execute.

#### Algorithm 13 SPLITTER(Builder, N.requests[])

```plaintext
1: for i = 0 to N - 1 do
2:    kaapi_request_reply(request[i], entrypoint, Builder.getCoPrimeGenerator());
3: end for
```

The coPrimeGenerator depends on the Builder type and allows the thief to generate a sequence of moduli. For instance the coPrimeGenerator for the earliest termination contains at one point a single modulus \(M\) which is returned by the next call of nextCoPrime() by the Builder.

The splitter function knows the number \(N\) of thieves that are trying to steal work to the same victim. Therefore it allows for a better balance of the work load. This feature is unique to KAAPI when compared to other tools having a work-stealing scheduler.

### 5.3 Synchronization

Now, the victim periodically tests the global termination of the computation (line 9 of algorithm 12). Depending on the chosen termination method (Early*CRA, etc.), the synchronization may occur at every iteration or after a certain number of iterations. The choice is made in order to e.g. amortize the cost of this synchronization or reduce the arithmetic cost of the reconstruction. Then each thief is preempted (line 11) and the code recovers its results before giving them to the Builder for future reconstruction (line 12).

The preemption operation is a two way communication between a victim and a thief: the victim may pass parameters and get data from one thief. Note that the preemption operation assumes cooperation with the thief code. The latter being responsible for polling incoming events at specific points (e.g. where the computational state is safe preemption-wise).

On the one hand, to amortize the cost of this synchronization, more primes should be given to the thieves. In the same way, the victim code works on a list of moduli inside the critical section (at line 3 returns a list of moduli, and at lines 5-6 the victim iterates over this list by repeatedly calling apply and update methods). On the other hand, to avoid long waits of the victim during preemption, each thief should test if it has been preempted to return quickly its results (see next section).

\(^5\)This depends on the implementation, most of the LinBox library functions are reentrant.
5.4 Thief entrypoint

Finally, algorithm 14 returns both the sequence of residues and the sequence of primes that were given to the BlackBox. This algorithm is very similar to algorithm 12.

Algorithm 14 Thief’s ENTRYPoint(M)
1: Builder.initialize();
2: list of v.clear();
3: list of p.clear();
4: while Builder.CoPrimePoint() not empty do
5: if kaapi_preemptPoint() then break; end if
6: p := Builder.nextCoPrime();
7: kaapi_stealbegin( splitter, Builder);
8: list of p.push_back(p);
9: list of v.push_back(BlackBox.apply(p));
10: kaapi_stealend();
11: end while
12: kaapi_stealreturn (list of v, list of p);

Lines 7 and 10 define a section of code that could be concurrent with steal requests. At line 5, the code tests if a preemption request has been posted by algorithm 12 at line 11. If this is the case, then the thief aborts any further computation and the result is only a partial set of the initial work allocated by the splitter function.

5.5 Efficiency

These parallel versions of the Chinese remainering have been implemented using KAAPI transparently from the LinBox library: one has just to change the sequential controller `cra-domain.h` to the parallel one.

In LinBox-1.1.7 some of the sequential algorithms which make use of some Chinese remainering are the determinant, the minimal/characteristic polynomial and the valence, see e.g. [20, 12, 11, 8] for more details.

We have performed these preliminary experiments on an 8 dual core machine (Opteron 875, 1MB L2 cache, 2.2Ghz, with 30GBytes of main memory). Each processor is attached to a memory bank and communicates to its neighbors via an hypertransport network. We used g++ 4.3.4 as C++ compiler and the Linux kernel was the 2.6.32 Debian distribution.

All timings are in seconds. In the following, we denote by $T_{seq}$ the time of the sequential execution and by $T_p$ the time of the parallel execution for $p = 8$ or $p = 16$ cores. All the matrices are from “Sparse Integer Matrix Collection” (SIMC)\(^6\).

Table 3 gives the performance of the parallel computation of the determinant for small invertible matrices (less than a second) and larger ones (an hour CPU) of the SIMC/SPG and SIMC/Trefethen collections. The small instance (ex-1) needed very few primes to reconstruct the solution. There, we can see the overhead of parallelism: this is due to some extra synchronizations and also to the large number of unnecessary modular comput-

\(^6\)http://ljk.imag.fr/CASYS/SIMC

<table>
<thead>
<tr>
<th>Matrix</th>
<th>d.r</th>
<th>$T_{seq}$</th>
<th>$T_{p=8}$</th>
<th>$T_{p=16}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>ex - 1</td>
<td>593, 2836</td>
<td>0.29[s]</td>
<td>0.16[s]</td>
<td>0.22[s]</td>
</tr>
<tr>
<td>ex - 3</td>
<td>5200, 71760</td>
<td>837.80[184]</td>
<td>123.56[193]</td>
<td>77.99[193]</td>
</tr>
<tr>
<td>t - 1500</td>
<td>500, 2040</td>
<td>0.21[59]</td>
<td>0.046[63]</td>
<td>0.036[63]</td>
</tr>
<tr>
<td>t - 3000</td>
<td>500, 4678</td>
<td>2.52[38]</td>
<td>0.36[144]</td>
<td>0.24[144]</td>
</tr>
<tr>
<td>t - 5000</td>
<td>500, 8478</td>
<td>15.19[249]</td>
<td>2.05[257]</td>
<td>1.31[256]</td>
</tr>
<tr>
<td>t - 7000</td>
<td>700, 12654</td>
<td>52.59[967]</td>
<td>6.50[368]</td>
<td>4.19[371]</td>
</tr>
</tbody>
</table>

Table 4: Timings for the computation of the determinant. $d$ is the dimension of the matrix, $r$ the number of non-zero coefficients, $k$ is the mean number of primes observed for the Chinese remainering using $p$ cores.

We show on table 4 the corresponding speed-ups of table 3 compared with a naive approach using OpenMP: for $p$ the number available cores, launch the computations by blocks of $p$ iterations and test for termination after each block is completed. For large computations the speed-up is quite significant before realizing that early termination was needed. Despite this we do achieve some speed-up.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>ex - 1</th>
<th>ex - 3</th>
<th>t - 1500</th>
<th>t - 3000</th>
<th>t - 5000</th>
<th>t - 7000</th>
<th>t - 2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>1.38</td>
<td>10.66</td>
<td>2.10</td>
<td>8.52</td>
<td>11.29</td>
<td>12.55</td>
<td>12.48</td>
</tr>
<tr>
<td>Alg.12</td>
<td>1.35</td>
<td>10.74</td>
<td>5.78</td>
<td>10.52</td>
<td>11.56</td>
<td>12.55</td>
<td>12.59</td>
</tr>
</tbody>
</table>

Table 4: Speed-up using 16 cores of algorithm 12 compared to a naive approach with OpenMP

the same since the computation is largely dominant. For smaller instances we see the advantage of reducing the number of synchronizations. On e.g. multi-user environments the advantage should be even greater.

6. CONCLUSION

We have proposed a new data structure, the radix ladder, capable of managing several kinds of Chinese reconstructions while still enabling fast reconstruction.

Then, we have defined a new generic design for Chinese remainering schemes. It is summarized on figure 3. Its main feature is the definition of a builder interface in charge of the reconstruction. This interface is such that any of termination (deterministic, early terminated, distributed, etc.) can be handled by a CRA controller. It enables to define and test remainering strategies while being transparent to the higher level routines. Indeed we show that the Chinese remainering can just be a plug-in in any integer computation.

We also provide in LinBox-1.1.7 an implementation of the ladder, several implementations for different builders and a sequential controller. Then we tested the introduction of a parallel controller, written with KAAPI, without any modification of the LinBox library. The latter handles the difficult issue of distributed early termination and shows good performance on a SMP machine.

In parallel, some improvement could be made to the early termination strategy in particular when the BlackBox is
fast compared to the reconstruction and when balanced and amortized techniques are required. Also, output sensitive early termination is very useful for rational reconstruction, see e.g. [21] and thus the latter should benefit from this kind of design.

7. REFERENCES


