Implementing Data Parallel Rational Multiple-Residue Arithmetic in Eden*

Extended and revisited version

Oleg Lobachev and Rita Loogen

Philipps–Universität Marburg, Fachbereich Mathematik und Informatik
Hans–Meerwein–Straße, D–35032 Marburg, Germany
{lobachev,loogen}@informatik.uni-marburg.de

Abstract. Residue systems present a well-known way to reduce computation cost for symbolic computation. However most residue systems are implemented for integers or polynomials. This work combines two known results in a novel manner. Firstly, it lifts an integral residue system to fractions. Secondly, it generalises a single-residue system to a multiple-residue one. Combined, a rational multi-residue system emerges. Due to the independent manner of single "parts" of the system, this work enables progress in parallel computing. We present a complete implementation of the arithmetic in the parallel `Haskell` extension `Eden`. The parallelisation utilises algorithmic skeletons. We compare our approach with `Maple`. A non-trivial example computation is also supplied.
Keywords: residue system, rational reconstruction, EEA, CRT, homomorphism, parallelisation, functional programming, parallel functional software, implementation report

1 Introduction

A common approach to reduce computation complexity of symbolic computations, i.e., intermediate expression swell, is a residue arithmetic. We regard residues of integers in this paper. A residue system w.r.t. some prime m is the field $\mathbb{Z} \setminus \langle m \rangle$. The addition and multiplication are the usual operations in $\mathbb{Z}$, combined with an integer division by m. The actual result is the residue of the division. The focus of this paper lies on systems with a) using multiple residues at the same time, b) capable of representing fractions. If we have an *a priori* upper bound on the result of our computation, then we can perform it in a certain residue class with the result remaining exact. The benefit is reduced computation time, especially for intermediate expressions, which might be significantly larger than the bound [18]. Using the Chinese Residue Theorem we can split a *single* large residue class into multiple smaller residue classes. The latter can be designed to fit into a machine word, making the single operation in a small residue class a constant time one. Thus we can obtain arbitrary precision by increasing the numbers of small residue classes.

* Supported by DFG grant LO 630-3/1.

```
farmBase :: Int → (Int → [a] → [[a]])      --^ n, distribute
                → ([[b]] → [b])            --^ combine
                → ([a] → [b])              --^ worker function
                → [a] → [b]                --^ what to do
farmBase np distr combine f tasks = combine $ parMapBase f $ distr np tasks

farm :: (a → b) → [a] → [b]
farm  = (farmBase noPe unshuffleN shuffleN) ∘ map
```

Fig. 1. The implementation of `farm` in `Eden`. Type context for transmissible data is omitted.

The approach mentioned above is traditionally implemented for *integer* arithmetic. However, it is possible to represent certain subsets of *rational* numbers as integers in a residue class—and to recover the rational numbers from integers. This property holds for a bound on the input rational numbers and output residue class [8].

Eden This paper presents a parallel `Haskell` programmer's approach to a *rational multiple-residue arithmetic*. We develop it in the broader context of the SPICA project[1], an implementation of selected computer algebra algorithms using novel parallelisation techniques, i.e., algorithmic skeletons. Such skeletons implement common patterns of parallel computation like process farms, divide-&-conquer schemes, etc. The source code is written in `Haskell` [13] with GHC extensions. The parallelism constructs reside in a controlled subset of the code base. It is written in the parallel `Haskell` extension called `Eden` [12]. The latter incorporates explicit process creation and implicit communication. An *algorithmic skeleton* library is available for `Eden` [11]. Contrary to the typical imperative approach, the skeletons are implemented in `Eden` itself, as they are just higher-order functions. The skeleton library contains, for instance, a parallel process `farm`, implemented as a statically load-balanced parallel `map`, *cf.* Figure 1. This function applies its first argument which is a function to each element of its second argument which is a list. The farm skeleton divides the input list into packages of almost equal size which are processed in parallel. Using only skeleton calls for parallelism, we can refrain from using parallel primitives in `Eden` programs. `Eden` is implemented as a distributed memory language, but it performs also well on multicore SMP machines. `Haskell` and thus `Eden` is a statically typed language with polymorphism. Each language object has a type, obtained with Hindley-Milner type inference. We denote the type with `object :: type`.

Plan of the Paper The next section presents the rational-to-integer and integer-to-rational mappings. Section 3 presents in short an integer multiple-residue arithmetic. Section 4 reports on conversion of integer multiple-residue representation back to integers. Section 5 is the actual focus of the paper. Subsections 5.1 and 5.2 present the rational multiple-residue arithmetic, whereas Subsection 5.3

[1] `http://www.mathematik.uni-marburg.de/~lobachev/`

describes our approach to parallelism. Section 6 presents an example implementation using the arithmetic. Section 7 presents related work. Section 8 concludes.

Acknowledgements We would like to thank Tomas Sauer for supervising the first author's diploma thesis on a similar topic [10]. This paper's theoretical background relies heavily on this thesis and on an excellent book by Gregory and Krishnamurthy [8]. Previous version of this paper was presented and discussed at CASC'2010.

2 From Fractions to Integers and Back

Definition 1 (Residues and division). *We denote integer division with* $a = cm + r$ *as* $c = a \operatorname{div} m$ *and* $r = a \bmod m$. *The latter forms* a residue class *for given* m. *We define* $(\mathbb{Z}_m, \oplus, \odot) := (\mathbb{Z}, +, \cdot)/\langle m \rangle$. *If* m *prime,* $(\mathbb{Z}_m, \oplus, \odot)$ *is a field with* $x \circledcirc y = (x \circ y) \bmod m$ *for* $x, y \in \mathbb{Z}_m$ *and* $\circ \in \{+, -, \cdot, /\}$. *An element of* $\mathbb{Z}_m = \{0, \dots, m-1\}$, *denoted with* $|a|_m$, *is* $a \bmod m$. *With a small abuse of notation, we write in further simply* $\mathbb{Z}_m$ *for* $(\mathbb{Z}_m, \oplus, \odot)$. *Further, if in the division above* $r = 0$, *then we write* $m \mid a$, *else* $m \nmid a$. *A residue class modulo* multiple *residues* $\beta = [m_1, m_2, \dots, m_n]$ *is defined as* $\mathbb{Z}_\beta := \mathbb{Z}_{m_1} \times \ldots \times \mathbb{Z}_{m_n}$. *The corresponding arithmetic operations will be defined later.*

The well-known notion of an *extended euclidean algorithm* (EEA) in a matrix-vector form can be represented as follows.

Algorithm 1 (Standard EEA).
Input: seed matrix $\begin{pmatrix} a_1 & b_1 \\ a_2 & b_2 \end{pmatrix}$.

1. *If* $a_2 = 0$, *return* $[a_1, b_1]$.
2. *Let* $t = a_1 \operatorname{div} a_2$. *Set pairwise* $(a_1, a_2) \leftarrow (a_2, a_1 - ta_2)$ *and* $(b_1, b_2) \leftarrow (b_2, b_1 - tb_2)$. *Go to step 1.*

Output: $[a_1, b_1]$.

We implement Algorithm 1 in Figure 2, see top of the figure. The 2×2 matrix is represented by a nested pair of pairs. The `Integral` type class is a `Haskell` notion for ring $\mathbb{Z}$, which abstracts from the integer representation.

Definition 2. *We define the residue-based representation of the rationals* $|a/b|_m$ *as elements of* $\hat{\mathbb{Q}}_m$ *per [8]. The elements of* $\hat{\mathbb{Q}}_m$ *are integers in the* m*-modular residue arithmetic. The notion of* $|a/b|_m$ *stands for an integer modulo* m, *congruent to* $|a|_m \odot |b^{-1}|_m$. *Here* $\odot$ *denotes the multiplication modulo* m.

We can compute $|a/b|_m$ efficiently, using EEA.

Algorithm 2 (Rational-to-integer mapping).
Input: A fraction a/b, *an integer* m *with* $m \nmid a$, $m \nmid b$.
Start Algorithm 1 with input matrix

$$\begin{pmatrix} m & 0 \\ b & a \end{pmatrix}.$$

```
eeaStep :: (Integral a) ⇒ ((a, a), (a, a)) → ((a, a), (a, a))
eeaStep ((a1, a2), (b1, b2)) = ((a2, a3), (b2, b3))
    where t = a1 `div` a2
          a3 = a1 `mod` a2
          b3 = b1 - t*b2

eea :: (Integral a) ⇒ ((a, a), (a, a)) → ((a, a), (a, a))
eea ((a1, a2), (b1, b2))
  | a2 == 0 = ((a1, a2), (b1, b2))
  | otherwise = eea $ eeaStep ((a1, a2), (b1, b2))
```

```
convertFraction :: (Integral i) ⇒ Fraction i → i → Mod i
convertFraction (F x y) p
  = let ((d,_),(r,_)) = eea ((p,y),(0,x))
    in if d ≠ 1 then error "convertFraction" else makeZ r p
-- Type Mod i and function makeZ are explained in Figure 3.
```

```
eeaSearch :: (Integral a) ⇒ ((a, a), (a, a)) → a → Maybe (a, a)
eeaSearch ((a1, a2), (b1, b2)) n
  | a2==0 = Nothing
  | a2≠0 ∧ ¬(criteria a2 b2 n)
    = flip eeaSearch n $ eeaStep ((a1, a2), (b1, b2))
  | otherwise = Just (a2, b2)
  where criteria x y n = abs x < n ∧ abs y < n

restoreFraction :: (Integral i) ⇒ i → i → Maybe (i, i)
restoreFraction a m = eeaSearch ((a, m), (0, 1)) n
  where n = nFromM m -- converts m to n per (1), see Figure 3.
```

Fig. 2. A generic (Algorithm 1) and two special (Algorithms 2 and 3) implementations of extended euclidean algorithm in `Eden`.

Return the second element of the output vector.
Output: an integer, representing $|a/b|_m$.

Algorithm 2 returns the desired *recoverable* result if a *bound* on m and a/b holds. It is rigorously discussed [7,19,20,9,8]. We summarise.

Definition 3 (Farey fractions). *All vulgar fractions* a/b *satisfying* $|a| \le N$, $|b| \le N$ *are called* Farey fractions *of order* N.

Proposition 4. *If*

$$N \le \sqrt{\frac{m}{2}} \tag{1}$$

holds, it is possible to recover the original Farey fraction a/b *of order* N *from integer* $|a/b|_m$.

Algorithm 3 (Integer to Farey fraction).
Input: an integer $x = |a/b|_m$, m.

1. *Compute* N *from* m *per* (1).
2. *Start Algorithm 1 with seed matrix*

$$\begin{pmatrix} m & 0 \\ x & 1 \end{pmatrix}.$$

```
data Mod a = Z a a
makeZ :: Integral a ⇒ a → a → Mod a
lift2z :: Integral a ⇒ (a → a → a) → Mod a → Mod a → Mod a
lift2z f (Z a p) (Z b q)
  | p ≠ q = error "Different residue classes!"
  | otherwise = makeZ (f a b) p

-- P.+ denotes (+) instances for Integral type class
-- from the Prelude. It corresponds to the ring ℤ.
instance (Integral a) ⇒ Num (Mod a) where
    (+) = lift2z (P.+)
    (−) = lift2z (P.-)
    (∗) = lift2z (P.∗)
instance (Integral a) ⇒ Fractional (Mod a) where
    (/) (Z a p) (Z b q) = -- use EEA
```

Fig. 3. Required function types for single-residue arithmetic in `Eden`.

```
type IMods a = [Mod a]
makeIZ′ :: (Integral a, Integral b) ⇒ a → [a] → IMods b
makeIZ′ value primes = map (makeZ value) primes
instance (Integral a) ⇒ Num (IMods a) where
    (+) = zipWith (+)
    -- and so on...
```

Fig. 4. Implementing the multiple-residue integer arithmetic.

3. *In each step of the algorithm check, whether* $|a_1|$ *and* $|b_1|$ *are both* $\leq N$*. If so, return the fraction* b_1/a_1 *(sic). If Algorithm 1 terminates without producing such a pair of numbers, fail.*

Output: either a Farey fraction a/b *or a failure.*

The correctness of Algorithm 3, the uniqueness of the fraction b_1/a_1, and the criteria for the input of the algorithm, needed to succeed, are proved in [8]. The first proof known to us is in [20]. We name the mapping "rational reconstruction" per [18]. We show `Eden` implementations of Algorithms 2 and 3 on Figure 2. Further, we need a way to refer to a single residue arithmetic in $\mathbb{Z}/\langle m\rangle = \mathbb{Z}_m$. The details are well-known, we present source code signatures in Figure 3.

3 An Integer-based Multiple-Residue Arithmetic

Let us consider a *multiple*-residue system $\mathbb{Z}_\beta$ with more than one residue. Hence, β is a vector. For the sake of simplicity we consider elements of β to be prime numbers. Then for $\beta = [m_1, m_2, \ldots, m_n]$ single residue classes are $\mathbb{Z}_{m_1}, \ldots, \mathbb{Z}_{m_n}$. With $M = m_1 \cdots m_n$, it holds that

$$\mathbb{Z}_{m_1} \times \ldots \times \mathbb{Z}_{m_n} = \mathbb{Z}_\beta \cong \mathbb{Z}_M. \quad (2)$$

The equation (2), read from left to right, is widely known as Chinese Residue Theorem, which we abbreviate to CRT. There are many different proofs of the

CRT; some of the constructive ones allow the algorithmic construction of the "large" residue. We call such proofs *implementations* of CRT, the other name in the literature is "Chinese Residue Algorithm", *cf.* [18]. We show a known approach to it in Section 4.

Further (2) facilitates a background for forth and backwards mappings between $\mathbb{Z}_\beta$ and $\mathbb{Z}_M$ as well as for defining the arithmetic. We will present this known result with a notion from functional programming.

Definition 5 (Map function). *For all functions operating on single elements:* `f :: a → b`*, we define a function* `map`*, which takes as its arguments such* `f` *and a* collection *of type* `[a]` *of elements of type* `a`*. The function* `map` *applies* `f` *to each element of its input collection and combines the results of each such application to its output collection of type* `[b]`*. Hence,* `map` *has the type* `(a → b) → [a] → [b]` *and a partial application* `map f` *has the type* `[a] → [b]`*. So we write*

```
map :: (a → b) → [a] → [b]
map f xs = [ f x | x ∈ xs ]
```

Corollary 6 (ZipWith function). *We define a binary version of* `map`*.*

```
zipWith :: (a → b → c) → [a] → [b] → [c]
zipWith f xs ys = [ f x y | x ∈ xs | y ∈ ys ]
```

Now we can define all four integral multiple-residue arithmetic operation on $\mathbb{Z}_\beta$ as a kind of `map` of their single-residue counterparts. Because `map` applies `f` to each single element in an independent manner, such definition of a multiple-residue arithmetic underlines its strength for vectorisation. All of the computation within a single "residue element" can be done independent from other residue elements. This will be the basis for the parallel implementation in Subsection 5.3. The implementation of the arithmetic falls back to `zipWith`—a variant of `map` for binary functions. Hence, the type for the multiple-residue system is just a list of single-residues. See Figure 4 for details. Now we can sketch the following.

Algorithm 4 (To integral multiple-residue).
Input: vector of primes β, integer x with no common factors with elements of β.
Compute $|x|_{m_i}$ for all elements m_i of β.
Output: $|x|_\beta$

4 Mixed-Radix Representation

We use the *mixed-radix* representation to convert entries from $\mathbb{Z}_\beta$ to $\mathbb{Z}_M$. This approach is described in detail in [8].

Definition 7 (Mixed-radix representation). *For a representation of an integer x w.r.t. a base vector $\rho = [r_1, \ldots, r_{k-1}]$ write $\langle x\rangle_\rho = [d_0, \ldots, d_{k-1}]$ with $s = d_0 + d_1r_1 + d_2r_1r_2 + \ldots + d_{k-1}r_1r2\cdots r_{k-1}$. Naturally, $0 \le d_i < r_{i+1}$.*

It is easy to prove that a mixed-radix representation is unique using repeated division with a remainder. The intriguing case is $\rho = \beta$. Thus we can obtain a mixed-radix representation $\langle x \rangle_\beta$ to a given multiple-residue representation $|x|_\beta$. Then we can convert the mixed-radix representation to $|x|_\mathbb{M}$ with $M = \prod m_i$ for $\beta = [m_1, \ldots, m_n]$. But we need to find a mixed-radix representation first. Denote with $\beta_{\{i,\ldots,n\}}$ a reduced vector $[m_i, \ldots, m_n]$. Further notation: t_k is the k-th element of the vector t, $t^{(k)}$ is the whole k-th vector t.

Algorithm 5 (To mixed-radix representation).
Input: $|x|_\beta$.

1. *Set* $t^{(1)} \leftarrow |x|_\beta$ *and* $i \leftarrow 1$. *Let* $d_0 \leftarrow t_1^{(1)}$ *and* n *is the length of* β.
2. *For* $i < n$ *do the following.*

$$t^{(i+1)} \leftarrow \left| \frac{t^{(i)} - |d_{i-1}|_{\beta_{\{i+1,\ldots,n\}}}}{m_i} \right|_{\beta_{\{i+1,\ldots,n\}}}$$
$$d_i \leftarrow t_{i+1}^{(i+1)}$$
$$i \leftarrow i + 1$$

Output: $\langle x \rangle_\beta = \langle d_0, \ldots, d_{n-1} \rangle$.

The length of t_i is reducing in each iteration step. It is now immediately clear how to compute $s = |x|_M$ from $\langle x \rangle_\beta$.

Algorithm 6 (From mixed-radix representation to an integer).
Input: $\langle x \rangle_\beta$

1. *Set* $s_1 \leftarrow d_{n-1}$
2. *For* $i = 2$ *to* $i \leq n$ *compute* $s_i \leftarrow d_{n-i} + s_{i-1} m_{n-i+1}$.

Output: $|x|_M = s_n$.

Algorithm 7 (From integral multiple-residue to an integer).
Input: $|x|_\beta$.

1. *Compute* $\langle x \rangle_\beta$ *with Algorithm 5.*
2. *Compute* $|x|_M$ *from* $\langle x \rangle_\beta$ *with Algorithm 6.*

Output: $|x|_M$.

Our implementation is presented in Figure 5. Other implementations of CRT are widely known and can be found in [3,18].

5 Rational Multiple-Residue System

5.1 The Mappings

Definition 8 (Elements). *We define an element of a rational multiple-residue system* $\mathbb{W}_\beta$ *as follows. Let* n *be the length of prime list* β. *Then such element is a list of pairs*

$$[(u_i, v_i) : i = 1, \ldots, n] \, .$$

```haskell
data SingleRadix a = MR a a
                     deriving (Eq, Show)
type MixedRadix a = [SingleRadix a]

-- ommited because of simplicity
valuesPrimes :: Integral a ⇒ IMods a → [(a, a)]
valuesRadices :: Integral a ⇒ MixedRadix a → [(a, a)]
getPrimes :: Integral a ⇒ IMods a → [a]
takeFirstValue :: (Integral a) ⇒ IMods a → a

restoreIZ :: Integral a ⇒ IMods a → Z a
-- Wrapper around restoreIZ'. In fact we restore Z a!
restoreIZ' :: Integral a ⇒ IMods a → a
restoreIZ' input = let (values, primes) = unzip $ valuesPrimes input
                   in convertMixedRadix $ mixedRadix input

inverses :: (Integral i) ⇒ i → [i] → IMods i
inverses k ps = flip makeIZ ps $ map (inverse k) ps

-- P.div is `div` from standard Prelude
lagrangians ps = let bigP = product ps
                 in map ((P.div) bigP) ps

-- a single step of mixed radix algorithm
mixedRadixStep :: Integral a ⇒ IMods a → IMods a
mixedRadixStep input
  = let (values, primes) = unzip $ valuesPrimes input
        h = head values
        diffs = (tail input) - (makeIZ' h $ tail primes)
        lagranges = inverses (head primes) (tail primes)
    in diffs * lagranges

mixedRadix :: Integral a ⇒ IMods a → MixedRadix a
mixedRadix input = zipWith (λb v → MR v b) (getPrimes input)
                   $ map takeFirstValue
                   $ takeWhile (λx → length x > 0)
                   $ iterate mixedRadixStep input

convertMixedRadix :: Integral a ⇒ MixedRadix a → Mod a
convertMixedRadix mixed
  = let residue = product primes
        (values, primes) = unzip $ valuesRadices mixed
        primes' = 1:primes
        -- (P.+) is (+) from standard Prelude, same with (*)
        conv (v, p) acc = (P.*) ((P.+) acc v) p  -- (acc + v) * p
    in flip makeZ residue $ foldr conv 0 $ zip values primes'
```

Fig. 5. Converting from $\mathbb{Z}_\beta$ to $\mathbb{Z}_M$ with mixed-radix algorithm.

```
data FSingleMod a = FM (Mod a) a
type FMods a = [FSingleMod a]

nFromM, mFromN :: Integral i ⇒ i → i
-- convert M to n per (1)
makeFZ :: Integral i ⇒ Fraction i → FMods i
-- forth mapping, see Figure 7
restoreFZ :: Integral i ⇒ FMods i → Maybe (Fraction i)
-- backwards mapping, see Figure 8
```

Fig. 6. Basic outline of rational multiple-residue implementation in `Eden`.

Here the components u_i are the residues and v_i are the powers of corresponding elements of β.

The implementation is in Figure 6. The authors of [8] define a similar residue system, we call $\mathbb{M}_\beta$ here. Its elements are similar to those of $\mathbb{W}_\beta$ and the arithmetical operations definitions coincide. The major difference lies in how the forth and backwards mappings are defined. Noteworthy, [7,9] define a yet another residue system, which differs from both $\mathbb{M}_\beta$ and $\mathbb{W}_\beta$ in not separating out the powers of primes v_i from the residues u_i. It is the predecessor of $\mathbb{M}_\beta$.

Given a fraction a/b and $\beta = [m_1, m_2, \ldots, m_n]$, satisfying (1) and (3), we can state the following.

Algorithm 8 (Common outline of forth mapping).
Input: fraction a/b, residues $\beta = [m_1, \ldots, m_n]$.

1. *Extract common factors v_i of all m_i and a/b. Remember $v_1, \ldots, v_n$.*
2. *Convert the resulting fraction to an integer modulo $M = m_1 \cdots m_n$ with Algorithm 2.*
3. *Convert the resulting integer to a multiple-residue system modulo β with Algorithm 4. Store the results in a list $[u_1, \ldots, u_n]$.*

Output: rational multiple-residue representation of a/b being $[(u_1, v_1), \ldots, (u_n, v_n)]$.

The key difference between our approach and $\mathbb{M}_\beta$ is in

$$\frac{a^{(1)}}{b^{(1)}} = \frac{a}{b} m_1^{v_1} \quad \frac{a^{(2)}}{b^{(2)}} = \frac{a}{b} m_2^{v_2} \quad \ldots \quad \frac{a^{(n)}}{b^{(n)}} = \frac{a}{b} m_n^{v_n}. \tag{3}$$

With these equations, the forth mapping for $\mathbb{W}_\beta$ takes in step 2 the value $a^{(i)}/b^{(i)}$ for i-th residue class in the system. The forth mapping for $\mathbb{M}_\beta$ extracts *all* factors from the input fraction for all residue classes. Unfortunately, this leads to an instable addition. Our approach does not have such a problem. We will show an example, underlying the difference of both approaches after the definition of arithmetic operations as Example 13 on page 12.

Algorithm 9 (Forth mapping).
Input: fraction a/b, residues $\beta = [m_1, \ldots, m_n]$.

```haskell
detectPower :: (Integral i, Num n) ⇒ i → i → (n, i)
-- Code omitted. Example: detectPower 40 2 = (3, 5)

convertFraction :: (Integral i) ⇒ i → i → i → Mod i
-- Code omitted. Converts fraction to integer modulo m

extractFactors :: (Integral i, Num n) ⇒ i → [i] → [(n, i)]
extractFactors x ps = map (detectPower x) ps

makeFZ' :: Integral i ⇒ i → i → [i] → FMods i
makeFZ' a b ps | gcd a b == 1
  = let (ws, ys) = unzip $ extractFactors a ps
        (qs, zs) = unzip $ extractFactors b ps
        vs = zipWith (−) ws qs -- well-defined
        cs = zipWith3 convertFraction ys zs ps
    in zipWith FM (cs) vs
               | otherwise = -- recursive call

makeFZ :: Integral i ⇒ Fraction i → FMods i
makeFZ = -- a trivial constructor expansion
```

Fig. 7. Forward mapping (Algorithm 9).

1. *Extract common factors* v_i *from* a/b *per* (3). *This results in* $v_1, \ldots, v_n$ *and* $a^{(1)}/b^{(1)}, \ldots, a^{(n)}/b^{(n)}$.
2. *For* $i \in \{1, \ldots, n\}$ *convert each* $a^{(i)}/b^{(i)}$ *to a value* u_i *modulo* m_i *with Algorithm 2.*

Output: an element $[(u_1, v_1), \ldots, (u_n, v_n)]$ *of* $\mathbb{W}_\beta$*, being rational multiple-residue representation of* a/b*.*

The implementation of this algorithm is on Figure 7. Note that in Algorithm 9 we convert each fraction $a^{(i)}/b^{(i)}$ separately, resulting in up to n calls of Algorithm 2. The backward mapping is defined as follows.

Algorithm 10 (Backward mapping).
Input: $[(u_1, v_1), \ldots, (u_n, v_n)] \in \mathbb{W}_\beta$, $\beta = [m_1, \ldots, m_n]$

1. *Compute* $M = m_1 \cdots m_n$ *and* $N = \sqrt{M/2}$.
2. *Compute* $a'/b' = m_1^{v_1} \cdots m_n^{v_n}$.
3. *For* $i \in \{1, \ldots, n\}$ *distort the values of* u_i*. Let*

$$\hat{u}_i := u_i / \prod_{j \neq i} m_j^{v_j}.$$

4. *Regard* $[\hat{u}_1, \ldots, \hat{u}_n]$ *an integer multiple-residue value in* $\mathbb{Z}_\beta$*. Find its representation* q *in* $\mathbb{Z}_M$ *with an implementation of CRT (Algorithm 7).*
5. *Find fraction* a/b *of order* N*, such that* $|a/b|_M = q$ *with Algorithm 3. If it succeeds, continue. Else fail.*

Output: aa'/bb' *or failure.*

Implementation of the latter algorithm is presented on Figure 8. How does the input set of Algorithm 9 look like? This set consists of Farey fractions of corresponding order N *and* their products with powers $v_1, \ldots, v_n$ of $m_1, \ldots, m_n$. For

```
getM :: Integral i ⇒ FMods i → Integer
-- returns the product of all primes in the system

stripPowers :: Integral i ⇒ FMods i → (i, i, FMods i)
-- set v_i = 0 for all i and compensate

restoreFZ' :: Integral i ⇒ FMods i → (Maybe (i,i), (i,i))
restoreFZ' x = let m = getM x
                   n = nFromM m
                   (nom, denom, strips) = stripPowers x
                   z = convertToIntResidues strips
                   r = toIntegral $ restoreIZ' z
                   e = eeaSearch ((m, r), (0, 1)) n
               in (e, (nom, denom))

restoreFZ :: Integral i ⇒ FMods i → Maybe (Fraction i)
restoreFZ = -- compute in Maybe monad the product of fraction e with nom/denom
```

Fig. 8. The outline of the backwards mapping (Algorithm 10).

the restricted values of v_i the shape of this set is shortly discussed in [10]. If we do not restrict the values of v_i, then it is infinite. Further questions on the shape of this set are open.

5.2 The Arithmetic

Now we have to define the actual arithmetic on $\mathbb{W}_\beta$. Each operation is defined for a single residue (use **map**! The rational system is still independent in its components). These definitions coincide with ones for $\mathbb{M}_\beta$ from [8], but not with ones from [7]. We begin with the definition of multiplication, since it is the simplest operation in the system.

Definition 9 (Multiplication). *The product of* (u, v) *and* (μ, ν) *modulo* m *is defined as* $(|u\mu|_m, v + \nu)$.

The implementation is straightforward:

```
(FM u1 v1) * (FM u2 v2) = FM (u1*u2) (v1+v2)
```

Definition 10 (Multiplicative Inverse). *The inverse of* (u, v) *is* $(|u|_m^{-1}, -v)$.

Note, it is easy and well-known, how to compute $|u|_m^{-1}$, the multiplicative inverse of u modulo m with EEA, Algorithm 1, for such u and m, that $\gcd(u, m) = 1$. It coincides with computing an integer representation of a fraction $1/u$ with Algorithm 2, a standard approach in residue rings. The sum of (u, v) and (μ, ν) modulo m is $(|u + \mu|_m, v)$ if $v = \nu$ and just (u, v) for $|v| < |\nu|$ with a single exception for sum of something with zero being the non-zero summand, regardless of the power of m. A more formal definition follows.

Definition 11 (Addition). *The sum of* (u, v) *and* (μ, ν) *modulo* m *is defined as follows. Let* $u \oplus \mu = |u + \mu|_m$. *We write in this table* v *for positive values,* $-v$

for negative and 0 *for zero.*

$+$	$(0,z)$	(u,v)	$(u,0)$	$(u,-v)$
$(0,\zeta)$	$(0,0)$	(u,v)	$(u,0)$	$(u,-v)$
(μ,ν)	(μ,ν)	A	$(u,0)$	$(u,-v)$
$(\mu,0)$	$(\mu,0)$	$(\mu,0)$	$(u\oplus\mu,0)$	$(u,-v)$
$(\mu,-\nu)$	$(\mu,-\nu)$	$(\mu,-\nu)$	$(\mu,-\nu)$	B

The two subcases are:

$$A=\begin{cases}(u,v) & \text{if } v<\nu\\ (u\oplus\mu,v) & \text{if } v=\nu\\ (\mu,\nu) & \text{if } v>\nu\end{cases}\quad B=\begin{cases}(u,-v) & \text{if } -v<-\nu\\ (u\oplus\mu,v) & \text{if } v=\nu\\ (\mu,\nu) & \text{if } -v>-\nu\end{cases}$$

Further holds, $z,\zeta\in\mathbb{Z}$*. The zero element is not unique because of* $(0,z)$ *with* $z\neq 0$*, but we* norm *it to the standard representation* $(0,0)$*.*

Definition 12 (Additive Inverse). *The additive inverse of* (u,v) *modulo* m *is* $(|-u|_m,v)$*.*

The actual arithmetic operations on $\mathbb{W}_\beta$ are defined by lifting the above single-element operations with `zipWith` to lists:

```
instance (Integral a) ⇒ Num (FMods a) where
    (+) = zipWith (+)
    (−) = zipWith (−)
    (∗) = zipWith (∗)
instance (Integral a) ⇒ Fractional (FMods a) where
    (/) = zipWith (/)
```

The code for addition, the most complicated operation even for single-element inputs, is presented in Figure 9.

Now, given the arithmetic, we can show that our approach is better than $\mathbb{M}_\beta$ from [8]. Regard an example computation [10].

Example 13 (Counterexample for $\mathbb{M}_\beta$*).* Let $a=1/21$ and $b=1/3$. We compute in $\mathbb{M}_\beta$ modulo $\beta=[5,7,11,13]$. Per (1), all fractions of order 50 are on the safe side. As $\mathbb{M}_\beta$ needs to extract all factors of elements of β from all elements of the residue system, we obtain representations $[(2,0),(5,-1),(4,0),(9,0)]$ for a and $[(2,0),(5,0),(4,0),(9,0)]$ for b. The sum is $[(4,0),(5,-1),(8,0),(5,0)]$, we obtain $2/21$ as the result, contrary to the correct result $8/21$. The same example with $\mathbb{W}_\beta$ of the same scale results in $[(1,0),(5,-1),(10,0),(5,0)]$ for a and $[(2,0),(5,0),(4,0),(9,0)]$ for b. The sum is $[(3,0),(5,-1),(3,0),(1,0)]$, yielding the correct result $8/21$.

Theorem 14 (Well-definiteness). *The arithmetic operations in* $\mathbb{W}_\beta$ *produce correct results.*

Proof. We consider again the element-wise operations.

```haskell
instance (Integral a) ⇒ Num (FSingleMod a) where
    (+) x y = addSingle x y
    (−) x y = x + (additiveInverseSingle y)
    -- etc.
```

```haskell
addSingle :: (Integral a) ⇒ FSingleMod a → FSingleMod a → FSingleMod a
addSingle (FM (Z 0 p) _) (FM (Z 0 p′) _) | p==p′ = FM (Z 0 p) 0
addSingle (FM (Z 0 _) _) y = y
addSingle x (FM (Z 0 _) _) = x
addSingle (FM u 0) (FM u′ 0) = FM (u+u′) 0
addSingle (FM u v) (FM u′ 0)   | v >0  = FM u′ 0
                               | v <0  = FM u v
addSingle (FM u 0) (FM u′ v′) | v′>0  = FM u 0
                               | v′<0  = FM u′ v′
addSingle (FM u v) (FM u′ v′) | v<v′  = FM u v
                               | v>v′  = FM u′ v′
                               | v==v′ = FM (u+u′) v
addSingle _ _ = error "Bad case!" -- never happened
```

```haskell
additiveInverseSingle (FM (Z u p) v) = FM (Z (p-u) p) v
```

Fig. 9. Additive operations in a single fractional residue class.

1. The addition works, despite looking somewhat strange. Let $(|a/b|_m, v)$ and $(|\alpha/\beta|_m, \nu)$ be the summands. The trivial case for summation with zero is clear. The case of $v = \nu$ is also not endearing. All left is the complicated case $v \neq \nu$. Without loss of generality, let $|v| < |\nu|$. Now we have three non-trivial sub-cases for different signs of v and ν, all other cases can be seen as one of those with places swapped. Let us consider one of them: the case "$0 < v < \nu$".
$$\frac{am^v}{b} + \frac{\alpha m^\nu}{\beta} = \frac{am^v}{b} + \frac{\alpha m^{\nu - v} m^v}{\beta} = \frac{am^v}{b} + \frac{\alpha m^v}{\beta} m^{\nu - v}.$$
If we extract all we can, namely m^v, the summand with further factor of m turns into zero modulo m, we have exactly $(|a/b|_m, v)$ remaining. All other cases are analogue.
2. The additive inverse is correct. Changing the sign changes not the factors, thus no change at v. The addition of (u, v) and $(|-u|_m, v)$ returns $(0, v)$, which is zero.
3. The multiplication is straight-forward. It follows
$$\frac{a}{b} m^v \cdot \frac{\alpha}{\beta} m^\nu = \frac{a\alpha}{b\beta} m^{v+\nu},$$
which is exactly what we see.
4. The multiplicative inverse is also correct:
$$(u, v) \cdot (|u|_m^{-1}, -v) = (|u \cdot u^{-1}|_m, v - v) = (1, 0).$$

□

Remark. Note that $\mathbb{W}_\beta$ does not give any guarantee on the correctness of the result, if the latter does not satisfy the bound (1).

Theorem 15 (Correctness). *The algorithms 9 and 10 are correct.*

Proof (Sketch). Call φ the mapping, defined by Algorithm 9 and ψ the one of Algorithm 10. Let F_N be Farey fractions of order N, let $X \subset \mathbb{Q}$ be the domain of φ with codomain $\mathbb{W}_\beta$. It holds $F_N \subset X$, if (1) holds for N and $M = \prod \beta$. Theorem 14 essentially shows $\varphi(a \circ b) = \varphi(a) \circ \varphi(b)$ for $\circ \in \{+, \cdot\}$. It is easy to show that zero and unity are preserved. Hence, φ is a ring homomorphism. ✓

As for ψ, it is a bit more tricky. The mapping $\psi : \mathbb{W}_\beta \to \mathbb{Q}$ is partial. However, if we constrain it[2] to F_N in the domain: $\psi|_N : \mathbb{W}_\beta \to F_N$, where N and β are in the same relation as above, then $\psi|_N$ is total per Proposition 4. Because $\mathbb{Q} = \bigcup_{N \in \mathbb{N}} F_N$, if a proper scale β is chosen, a fraction of arbitrary size can be mapped back to $\mathbb{Q}$. ✓ †

5.3 Parallelism

Multiple-residue arithmetic is known for its data parallelism potential. We compute with different residues in a fully independent manner, without a need for a communication in-between. As our implementation of rational multiple-residue arithmetic conforms to this principle, we can immediately make a step from a (sequential) `Eden` implementation to (parallel) `Eden` code.

Suppose, we have some function `f :: FMods Int → FMods Int`. This function could be implemented in `Eden` as `f = map g`, where `g :: FSingleMod Int → FSingleMod Int`. It suffices to write `f = farm g` to obtain a parallel `Eden` implementation. The skeleton `farm` implements a parallel `map` behaviour and is part of `Eden`'s skeleton library.

A further advantage is provided by the `Eden` type system. As both `FMods` and `FSingleMod` are instances of the standard `Num` and `Fractional` type classes, we could use the standard arithmetical notation of $+, -, \cdot, /$ in the implementation of the function `g` from above. Even more: the generalised type of `g` is `g :: (Num a, Fractional a) ⇒ a → a`. This means, that we can use `g` for *any* arithmetic of our choice: be it the standard one, or the one presented above. In terms of computer algebra, one says that `g` is symbolic.

6 Testing the Arithmetic

In order to have a large enough task, we use matrix computations for testing the arithmetic. We choose the *LU* decomposition of matrices as our test problem.

6.1 Gauß Elimination

The idea of Gauß elimination is simple: we subtract multiples of one matrix column from another one to obtain zeroes beneath the matrix diagonal. Details

[2] In fact, the domain, on which ψ for a given β is total, seems to be some $Y \subset \mathbb{Q}$, with the evident assumption $X = Y$. However for the proof technique it is easier to limit ψ to $F_N \subset Y \subset \mathbb{Q}$.

```
-- matrix!(j, k) selects a single element of the matrix
gaussIterator (i, bound, matrix)
  = let ((ln, lm), (n, m)) = bound
        zs = [ ((j, k), makeEl i j k) | j ∈ [i+1..n], k ∈ [i..n] ]
        makeY (i, j) = matrix!(j, i) / matrix!(i, i)
        makeEl i j k = matrix!(j, k) - makeY(i, j) * matrix!(i, k)
    in (i+1, bound, matrix//zs)

gauss :: (Num x, Fractional x) ⇒ MatArr Int x → MatArr Int x
gauss matrix
  = let ((ln, lm), (n, m)) = bounds matrix
        (_, _, result) = last $ take (n-ln+1)
          $ iterate gaussIterator (ln, ((ln, lm), (n, m)), matrix)
    in result
```

Fig. 10. Gauß elimination in `Eden`.

on Gauß elimination can be found in any numerical linear algebra book, like the classic [6]. We use determinant computation via Gauß elimination as a test for our arithmetic.

However, contrary to common approaches, we perform the *exact* computation. The input matrix is filled with fractions, the determinant is also fractional. As an upper bound on determinant size for a given matrix exists, we can ensure our arithmetic is exact.

We implement matrices in `Eden` as arrays. This data type is not quite native for pure functional language, but we did not want to use nested lists for matrix representation. We write `type MatArr a b = Array (a, a) b` whereas the typical usage of this type would be `MatArr Int (Fraction Integer)` or `MatArr Int (FMods Int)`. The sequential determinant computation would be as simple as

```
det :: (Num x, Fractional x) ⇒ MatArr Int x → x
det = product ∘ diag ∘ gauss
```

with a trivial implementation of `diag :: (Ix a, Num b) ⇒ MatArr a b → [b]`. We omit here our implementation of `gauss`, it is really straight-forward. It has the type `(Num x, Fractional x) ⇒ MatArr Int x → MatArr Int x`. We do not use pivoting here. To obtain a parallel implementation, we use higher-order function `lift1`, described below in Subsection 6.2. The actual parallel implementation of parallel Gauß elimination and a subsequent determinant computation is presented in Figure 11.

6.2 Technical Details of Implementation

In order to be able to distribute the data parallel tasks across the PEs, we need to rotate the hypercube, being matrix over a list of the fractional residue classes.

A quite performance boost resulted from two design decisions.

- *Do not send unneeded data.* We have implemented a special transmission data type for diagonal matrices. We take the diagonal of the transformed

```
-- lift1 is defined in Figure 12
gaussResidue myMap = lift1 myMap gauss
detResidue = product ∘ diag ∘ gaussResidue
```

```
-- parMap and farm are standard Eden skeletons
parMap, farm :: (a → b) → [a] → [b]
-- two parallel invocations
detParMap = detResidue parMap
detFarm = detResidue farm
```

Fig. 11. Residue-based invocation of parallel Gauß elimination. Implementation of `detResidue` is slightly simplified.

matrix *before* the communication takes place, thus majorly reducing communication bottleneck in the direction "workers to master".

– *List chunking.* This issue is very technical. Supporting `Eden` library defines list communication in form of streams, however each list element is sent separately. This is inefficient of large lists. A common solution to this problem among `Eden` programmers is "list chunking"—reducing a list to a nested list for the sake of communication, thus sending multiple list elements in a single message. Our experiments have shown that 100 elements "chunking" provides best results in this particular case.

6.3 Test Results

We have implemented distributed determinant computation of permuted, scaled with 1/3 Pascal matrices, using the above approach. As a Pascal matrix is unimodular, the final result is always known. The arithmetic of a right scale always performed correctly in our tests. A visualisation of the parallel program execution with EdenTV [1] is depicted in Figure 13. In the diagram, the horizontal axis indicates the time, the vertical axis shows PEs. The bars show process activity over time. Multiple processes can be placed on one PE. The colours correspond to a traffic light: red • (dark grey in a black and white version) is blocked, i.e., waiting for input. Yellow • (light grey) is "runnable", but not running, typical causes are garbage collection and communication in progress. Green • (grey) stands for running.

The initial delay of 1.3–1.5 second is due to the generation of the input matrix. Probably, some part of this computation is also due to the boilerplate code for parallelisation. Each parallel residue computation, seen from 1.25 second to 2.5–2.6 second perform for about 1.3 seconds. Then, the needed results—only the diagonal of the whole matrix!—are sent back to the PE 1. After a very short post-processing phase the program terminates. The visualised run was performed on a eight-core Intel Xeon machine with a 2.5 GHz CPU and 16 GB RAM. We used no special memory management and fixed the message buffer at 2 MB pro PE. The input matrix size was 100×100, we used 8 residues with primes of size $\approx 5 \cdot 10^4$. The bound on fraction size was $\approx 10^{17}$, the determinant

```
type TransMat i n = ((i, i), [n])
type SparseDiagMat i n = ((i, i), [((i,i), n)])

toL :: (Ix i, Num n) ⇒ MatArr i n → TransMat i n
fromL :: (Ix i, Num i, Num n) ⇒ TransMat i n → MatArr i n

toSD :: (Ix i, Num n) ⇒ MatArr i n → SparseDiagMat i n
fromSD :: (Ix i, Num i, Num n) ⇒ SparseDiagMat i n → MatArr i (Maybe n)

liftL :: (Ix i, Num i, Num n) ⇒
         (MatArr i n → MatArr i n ) → TransMat i n → TransMat i n
liftL f = toL ∘ f ∘ fromL

liftLS ::  (Ix i, Num i, Num n) ⇒
           (MatArr i n → MatArr i n ) → TransMat i n → SparseDiagMat i n
liftLS = toSD ∘ f ∘ fromL
```

```
toResiduePrimes :: (Ix i, Integral n) ⇒
                   [n] → MatArr i (Fraction n) → [MatArr i (FSingleMod n)]
fromResidueMaybe :: (Ix i, Integral n) ⇒
                    [MatArr i (FSingleMod n)] →
MatArr i (Maybe (Fraction n))
```

```
-- a map implementation / working function / primes / input value
-- lift1 :: ((a → b) → [a] → [b]) → (c → d) → [n] → m1 → m2
lift1 mymap f = fromResidueMaybe ∘ map fromL ∘ mymap (liftL f)
                ∘ map toL ∘ toResiduePrimes
```

Fig. 12. The function `lift1` and supporting code signatures. We omit technical details here.

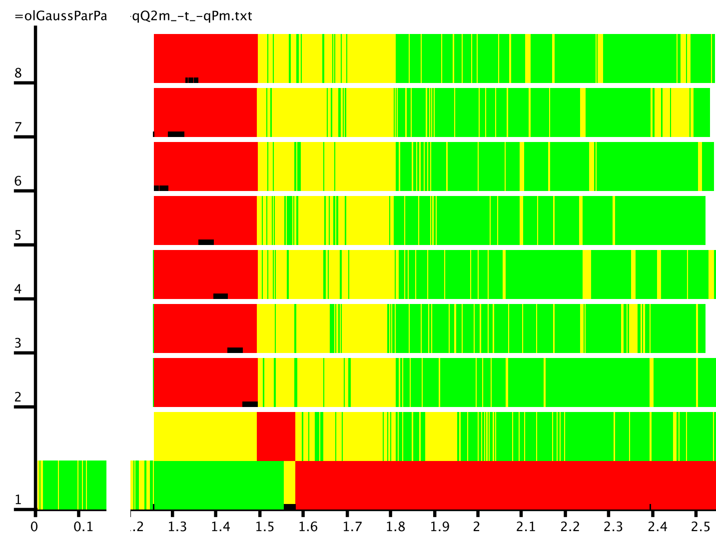

Fig. 13. A fragment of the runtime diagram of the test executable.

n	10	50	70	100
`Eden`, $\mathbb{W}_\beta$	0.0025	0.39	0.94	2.6
`Maple`, $\mathbb{Q}$	0.002	0.175	0.60	2.0

Table 1. Permuted scaled Pascal matrices. Comparison of our approach with `Maple`. Time is in seconds.

was a Farey fraction of order $\approx 5 \cdot 10^{47}$. We stress that the matrix operations' implementation is a prototype one. The essence of this example is not to make performance records, but to show that the arithmetic produces correct results in practice.

7 Related Work

7.1 Comparison with Maple

We compare our parallel implementation from above with a reference software: `Maple 13`. We compute the determinant of the same matrix using exact rational arithmetic and the determinant of the unscaled integer matrix using small single-residue arithmetic. In all cases the time measurement is incorporated in the software itself, we do not measure the time to generate the matrix, *cf.* Figure 13. We run all applications on the same hardware: the aforementioned eight core Intel machine.

The results are depicted in Table 1. We see that our `Eden` implementation is just a factor slower than optimised `Maple` implementation with standard fractions. So we have two different approaches to implement fractions mechanically, which are comparable in runtime. Still, our `Eden` measurement is based on a parallel execution on 8 PEs and `Maple` uses majorly only a single PE. The same `Eden` program on a single PE would run in 11.2 seconds. However, The time for the *single* residue-based computation in `Maple` is very low. Hence, solely the ability to compute the same result with a residue arithmetic would be a major advantage for `Maple`. This together with a possibility to execute the arithmetic in parallel on many cores would produce very high speedups. The reason for such improvement is reduced computational complexity, as described in [18]. We consider this issue next.

7.2 Speedups

For our `Eden` implementation we obtain the relative speedup of 4.31 and the absolute speedup is 3.34 on the same 8 PE machine as in previous sections. However, a parallel sparse Gauß elimination with parallel pivoting [4] produces for dense matrix input the speedup of ≈ 3.5 on 8 PE Cray C90 and ≈ 5 on 8 PE DEC AlphaServer 8400. Still for special sparse inputs the same algorithm on

Maple method				*Speedup*		
n	default	$\mathbb{Q}$	$\mathbb{Z}_m$	n	$\bar{s}$	$\hat{s}$
100	0.07	4.9	0.012	100	5.8	408
300	2.2	494	0.018	300	122	27378
1000	325	–	0.75	1000	433	–

Table 2. Random sparse matrices. Comparing `Maple` methods and stating upper bound on the speedups. Time is in seconds.

the latter machine achieves speedup of ≈ 7. Noteworthy, [4] describe an SMP implementation. Our approach works in a distributed memory setting.

As a further indicator of possible benefit, we compute in `Maple 13` determinants of larger random-generated matrices of small (≤ 1000) integers of density 1/2 using default, rational and residue-based implementations of built-in `Maple` function `Determinant`. We show the result in Table 2. The residue-based approach measures time for one residue class at $m = 1013$. Note, that the timings for the same matrix size differ for vulgar fraction computation in Tables 1 and 2. The values at $\bar{s}$ are the best possible speedup of default method and values at $\hat{s}$ show the best possible speedup of rational computation method, both provided ideal scalability of the multi-modular arithmetic. The latter assumes that computing p residues at p PEs takes as long as computing one residue at a single PE and that we have as many PEs as we need. Hence, this is the maximal possible speedup in the particular case. Of course practically measured values may and will vary. Still, we see a huge potential for using a multi-residue arithmetic for rational computations.

7.3 Further Related Work

Any decent system implements residue-based approach for reducing the intermediate expression swell. A lot of work on rational residue systems was done in [8] and preceding papers [7,19,20,9,14]. Another proof of Proposition 4 is in [15], [16] states an alternative divide-&-conquer algorithm. The earliest approaches to rational residues known to us are [17] and [2]. Our own work on this topic is [10].

8 Conclusions and Future Work

We have presented a rational multiple-residue arithmetic and its parallel implementation in `Eden`. We constructed a test case and provided a visualisation of the program execution. We present in this paper concise source code for almost all operations. The omitted parts are straightforward. Nevertheless, the complete source code package is available from the project homepage[3].

[3] `http://www.mathematik.uni-marburg.de/~lobachev/code/multimod/`

Further direction is the development of a parallel benchmark, based on the presented implementation, coupled with certain test input. Another point for future work would be an attempt for an adaptive computation. It would be interesting to see a `Maple` implementation of our approach. A distributed implementation, based on the remote data concept [5] should improve the speedup values of our implementation.

References

1. J. Berthold and R. Loogen. Visualizing Parallel Functional Program Executions: Case Studies with the Eden Trace Viewer. In *ParCo 2007*. IOS Press, 2007.
2. I. Borosh and A. S. Fraenkel. Exact solutions of linear equations with rational coefficients by congruence techniques. *Math. Comp.*, 20(93):107–112, 1966.
3. H. Cohen. *A Course in Computational Algebraic Number Theory*. Springer, 1995.
4. J. W. Demmel, J. R. Gilbert, and X. S. Li. An asynchronous parallel supernodal algorithm for sparse Gaussian elimination. *SIAM Journal on Matrix Analysis and Applications*, 1999.
5. M. Dieterle, T. Horstmeyer, and R. Loogen. Skeleton composition using remote data. In *Parallel Aspects of Declatative Languages (PADL)*, LNCS 5937, pages 73–87. Springer, Jan. 2010.
6. G. H. Golub and C. F. Van Loan. *Matrix computations*. Johns Hopkins Univ Pr, 1996.
7. R. T. Gregory. Error-free computation with rational numbers. *BIT Numerical Mathematics*, 21(2):194–202, 1981.
8. R. T. Gregory and E. V. Krishnamurthy. *Methods and Applications of Error–Free Computation*. Springer, 1984.
9. P. Kornerup and R. T. Gregory. Mapping integers and Hensel codes onto Farey fractions. *BIT Numerical Mathematics*, 23(1):9–20, 1983.
10. O. Lobachev. Multimodulare Arithmetik, Mar. 2007. Justus-Liebig-Universität Gießen. Diplomarbeit, in German. `http://www.mathematik.uni-marburg.de/~lobachev/diplom.pdf`.
11. R. Loogen, Y. Ortega-Mallén, R. Peña, S. Priebe, and F. Rubio. Parallelism Abstractions in Eden. In F. A. Rabhi and S. Gorlatch, editors, *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2003.
12. R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel Functional Programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.
13. S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, Dec. 2003.
14. T. M. Rao and R. T. Gregory. Conversion of Hensel codes to rational numbers. *Comp. Math.*, 10(2):185–189, 1984.
15. T. Sasaki and M. Sasaki. On integer-to-rational conversion algorithm. *ACM SIGSAM Bulletin*, 26(2):19–21, 1992.
16. T. Sasaki, Y. Takahashi, and T. Sugimoto. A divide-and-conquer method for integer-to-rational conversion. In *Symposium in Honor of Bruno Buchberger's 60th Birthday*, page 231, 2002.
17. A. Svoboda and M. Valach. Rational system of residue classes. *Stroje na Zpraccorani Informaci, Sbornik, Nakl. CSZV, Prague*, pages 9–37, 1957.
18. J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, second edition, 2003.

19. P. S. Wang. A p-adic algorithm for univariate partial fractions. In *Proc. ACM Symposium on Symbolic and Algebraic Computation*, pages 212–217. ACM, 1981.
20. P. S. Wang, M. J. T. Guy, and J. H. Davenport. p-adic reconstruction of rational numbers. *ACM SIGSAM Bulletin*, 16(2):3, 1982.