Parallel Implementation of Exact Matrix Computation Using Multiple *P*-adic Arithmetic

Xinkai Li and Chao Lu
Department of Computer & Information Sciences
Towson University, 8000 York Rd
Towson, MD 21252, USA
E-mail: clu@towson.edu

Jon A. Sjogren
AFOSR/InPharmex LLC, Suite 715
2308 Mt. Vernon Ave.
Alexandria, VA 22301, USA
E-mail: jsjogren8424@yahoo.com

Received 24 January 2013

Accepted 9 May 2013

Abstract

A P-adic Exact Scientific Computational Library (ESCL) for rational matrix operations has been developed over the past few years. The effort has been focusing on converting all rational number operations to integer calculation, and fully taking advantage of the fast integer multiplication of modern computer architectures. In this paper, we report our progress on parallel implementation of *P*-adic arithmetic by means of a multiple modulus rational system related to the Chinese remainder theorem. Experimental results are given to illustrate computational efficiency.

Keywords: Parallel computing, computational efficiency, error-free, *P*-adic, multiple modulus rational system, Chinese remainder theorem

1. Introduction

For the past several years, we have been developing *P*-adic Exact Scientific Computational Library (ESCL) for rational matrix operations. Based on Krishnamurthy [1, 2] and Dixon [3] theories, we have established a finite *P*-adic sequence calculation system [4-7]. But there is a problem that for certain complex matrix operations, even with small matrix sizes, the new method requires a long *P*-adic sequence to guarantee against overflow [6]. The longer the *P*-adic sequences are, the longer the calculation will take, and computational efficiency becomes an issue. One solution to this problem is to adopt parallel computing. It is difficult to realize parallel computation directly in *P*-adic arithmetic due to its data structure. If we combine the multiple modulus rational systems [8] and the *P*-adic arithmetic, then parallel computation can be realized, which was called multiple *P*-adic arithmetic by Morrison [9]. A similar idea was also mentioned by Limongelli, Loidl [10] and Koc [11]. This paper will be focused on parallel implementation of multiple *P*-adic arithmetic applied to rational matrices using *P*-adic exact computation. Overflow detection will also be addressed. Finally, comparison tests and experimental results will be presented.

2. Multiple Modulus Rational System (Extended Chinese Remainder Theorem)

2.1 *Chinese remainder theorem (CRT)*

Recalling the Theorem (Chinese remainder theorem) [2, 12], if $r \sim \{r_1, r_2, \cdots, r_s\}$is the residue representation of an integer r with respect to moduli$\{p_1, p_2, \cdots p_s\}$, where, $GCD(p_i, p_j) = 1$ for i ≠ j, define $p = \prod_{i=1}^{s} p_i$ and $p_i'$by $\frac{p}{p_i} p_i' \equiv 1 \, mod \, p_i$, then the solution of the system is given by

$$r \equiv \sum_{i=1}^{s} \frac{p}{p_i} p_i' r_i \mod p$$

If the given condition is $|r| < \frac{1}{2}p$, the value of r can be identified by:

$$r = \begin{cases} r & \text{if } r \leq (p-1)/2 \\ -(p-r) & \text{otherwise} \end{cases}$$

For example:

$$r \equiv 2 \bmod 3$$
$$r \equiv 3 \bmod 4$$
$$r \equiv 4 \bmod 5$$

According to the Chinese remainder theorem,

$$p = 60$$
$$p_1' = 2$$
$$p_2' = 3$$
$$p_3' = 3$$

$r \equiv 59 \, mod \, 60$
If given condition $|r| < \frac{1}{2}p$,

$$r = -1$$

2.2 *Extended Chinese remainder theorem to rational numbers [8]*

The Chinese remainder theorem deals with integers. It shows how to transform a large integer into sequence of small integers. There is also a way to transform a fractional number with large numerator and/or denominator into a sequence of small integers. This method has been named as multiple module number systems [8], which we like to call it the extended Chinese remainder theorem.

2.2.1. *How to calculate rational module*

For a rational number $\frac{b}{a}$ with $GCD(a, b) = 1$, the calculation of $\frac{b}{a} \, mod \, p$ ($p \geq 0, p \in$ Z) is defined as

$$r = ba' \bmod p \ (aa' \bmod p \equiv 1)$$

2.2.2. *How to decode from the extended Chinese remainder theorem [11]*

If $r \sim \{r_1, r_2, \cdots, r_s\}$ is the residue representation of a rational number r with respect to moduli $\{p_1, p_2, \cdots p_s\}$ where $GCD(p_i, p_j) = 1$ for $i \neq j$, then the decoding algorithm is given as follows.

Decoding algorithm

```
Step 1:  Chinese remainder theorem
         $p = \prod_{i=1}^{s} p_i$
         For $i = 1$ to $s$
         Using extended Euclidean algorithm
         to find $p_i'$ by $\frac{p}{p_i} p_i' \equiv 1 \, mod \, p_i$
         End
         $\bar{r} = \sum_{i=1}^{s} \frac{p}{p_i} p_i' r_i \, mod \, p$
Step 2:  Euclidean algorithm
         $u_{-1} = p, u_0 = \bar{r}$
         $v_{-1} = 0, v_0 = 1$
         $i = -1$
         While $u_i < \sqrt{p}$
                 $q_i = \lfloor u_{i-1}/u_i \rfloor$
                 $u_{i+1} = u_{i-1} - q_i u_i$
                 $v_{i+1} = v_{i-1} + q_i v_i$
                 $i++$
         End
Rational solution:
         $r = ((-1)^i u_i / v_i)$
```

2.2.3. *How to identify the bound of the representation of a fraction number from the extended Chinese remainder theorem*

The proof process will be the same as “Decoding to Rational Form” part in Dixon’s paper [3].
Define $r = \frac{a}{b}$, $GCD(a, b) = 1$ and $\delta = max(a, b)$, according to Dixon’s theory that if δ satisfies $\delta \leq \lambda\sqrt{p}$ *($\lambda = 0.618 \cdots$* is a root of $\lambda^2 + \lambda - 1 = 0$*)*, we can use the decoding algorithm to get the rational number back.
For example, we choose $r = \frac{1}{7}$ and $p_1 = 3, p_2 = 4, p_3 = 5$ to check the decoding process:

$$r \equiv 1 \bmod 3$$
$$r \equiv 3 \bmod 4$$
$$r \equiv 3 \bmod 5$$

Step 1:

Using the Chinese remainder theorem, we get,

$$p = 60, \overline{r} = 43$$

Step 2:

By the Euclidean algorithm, we get,

$$u_{-1} = 60, v_{-1} = 0$$
$$u_0 = 43, v_0 = 1$$
$$u_1 = 17, v_1 = 1$$
$$u_2 = 9, v_2 = 3$$
$$u_3 = 8, v_3 = 4$$
$$u_4 = 1, v_4 = 7$$

The rational solution is,

$$r = \frac{1}{7}$$

3. Implementation of the Extended Chinese Remainder Theorem with P-adic Arithmetic

By the nature of the extended Chinese remainder theorem, it can be implemented on parallel computers. The idea can be demonstrated as follows:

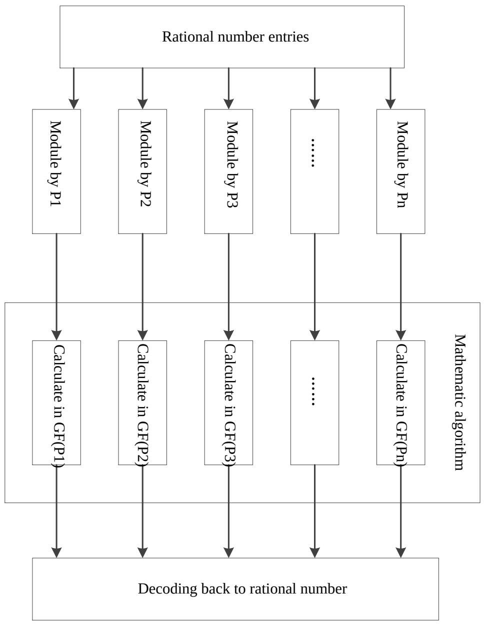


Fig. 1. *Extend CRT parallel implementation chart*

But in practice, there is a disadvantage of direct application. For a rational number $\frac{b}{a}$ and a prime p, if a and p are not relatively prime, we cannot get the result of $\frac{b}{a} \bmod p$. The way to solve this problem is to combine Hensel code calculation systems with the extended Chinese remainder theorem.

3.1. *P-adic (Hensel code) arithmetic [1, 4-6]*

Any rational number can be coded into P-adic sequence by the following algorithm. $\alpha = \frac{b}{a} \cdot p^n,\ a, b, n \in N, b \neq 0$, and $GCD(a, b)$, $GCD(a, p)$, $GCD(b, p) = 1$ can be written as $\propto = \sum_{i=k}^{\infty} a_i p^i, k \in N$.
The conversion process is the following:

Step 1. $\propto \bmod p = a_0$
Step 2. $\propto = (\propto - a_1)/p$, go to Step 1 to get a_1.
Continue Step 1 and Step 2, to get a_i
Finally, $\alpha = p^n \cdot \sum_{i=0}^{\infty} a_i p^i = \sum_{i=n}^{\infty} a_{i-n} p^i$

The P-adic sequence with point position n will have the following form:

$$a_n a_{n+1} \cdots a_{-2} a_{-1} . a_0 a_1 a_2 \cdots \qquad for\ n < 0$$
$$. a_0 a_1 a_2 \cdots \qquad for\ n = 0$$
$$.000 a_0 a_1 a_2 \cdots \qquad for\ n > 0$$

Conventionally, we write P-adic sequence as

$a_{i-n} a_{i-n+1} \cdots$ point position = n
point position means the position of a_0

For example, taking p = 3, for 1/5 and 2/5 the 5-adic expansions are,

$$\frac{1}{5} = .2012101210...$$

$$\frac{2}{5} = .1121012101...$$

3.1.1. *Addition/Subtraction*

The addition of P-adic is similar to the binary numeral addition. The difference is that P-adic addition process is calculating from left to right.
For example of computing $\frac{1}{6} + \frac{1}{2} = \frac{2}{3}$ for p = 5.

$$\frac{1}{6} = .140404040 \ldots \quad \frac{1}{2} = .32222222 \ldots$$

In the process, the position of point should be kept in alignment.

```
 .140404040···
 .322222222···
 ─────────────
 .413131313···
```

We can check that the 5-adic of

$$\frac{2}{3} = .413131313\ ...$$

Subtraction can be considered as addition,

$$\alpha - \beta = \propto + (-\beta)$$

3.1.2. *Multiplication/Division*

The multiplication of P-adic is also similar to the binary numeral multiplication. The difference is also that P-adic multiplication is calculating from left to right. The point position of the result equals to:

point1 + point2.

For example of $\frac{1}{3} \times \frac{1}{6} = \frac{1}{18}$ with $p = 5$:

$$\frac{1}{3} = .231313131\cdots$$

$$\frac{1}{6} = .140404040\cdots$$

The multiplication can be shown:

```
          .2313131313131 · · ·
        × .1404040404040 · · ·
------------------------------------------
          2313131313131 · · ·
            331313131313· · ·
              00000000000 · · ·
                3313131313 · · ·
                  000000000 · · ·
                    33131313 · · ·
                      0000000 · · ·
                        331313 · · ·
                          00000 · · ·
                            3313 · · ·
                              000 · · ·
                                33 · · ·
        +                         0· · ·
  ------------------------------------------------
--
            .2103341103341 · · ·
```

Check the result with 5-adic representation:

$$\frac{1}{18} = .21033411033411.....$$

Division can be carried out as a multiplication process. First we use recursive method to get the inverse of the dividend then do multiplication. The point position for division equals to:

point1 - point2.

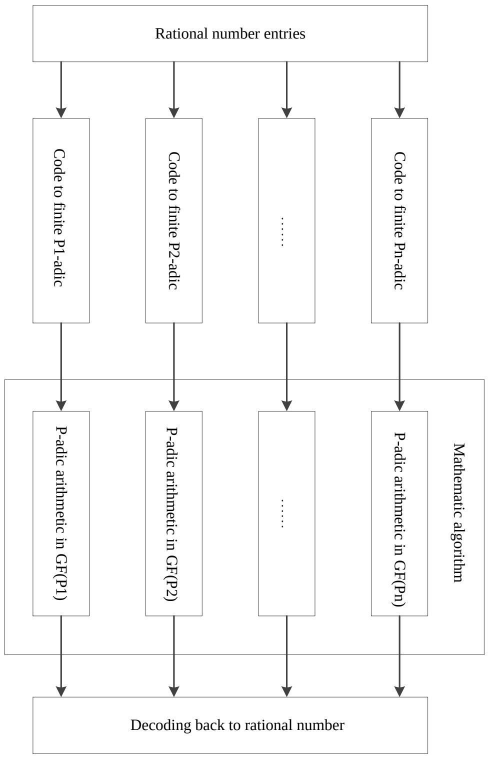
3.1.3. *Hensel code*

The encoded P-adic sequence is usually infinite. The way to choose a finite P-adic sequence used in exact rational computation is called Hensel code arithmetic [1]. The Hensel codes are closed with respect to basic arithmetic operations (ADD/SUBTRACT) and (MULTIPLY/DIVIDE).

For each Hensel code $H(p, r, \propto)$, p means the prime, r means the length of the P-adic sequence, $\propto$ means the finite P-adic sequence.

3.2. *Combining P-adic arithmetic with the extended Chinese remainder theorem*

P-adic arithmetic can be combined with the extended Chinese remainder theorem to do exact computing. It was called multiple P-adic algorithm [9]. In each $GF(p_i)$ we can use finite P-adic sequence to do calculation, the flow chart is the following:



Rational number entries
Code to finite P1-adic
Code to finite P2-adic
……
Code to finite Pn-adic
P-adic arithmetic in GF(P1)
P-adic arithmetic in GF(P2)
……
P-adic arithmetic in GF(Pn)
Mathematic algorithm
Decoding back to rational number

Fig. 2. *Extended CRT combined with P-adic arithmetic for parallel implementation*

The decoding process:
If $x \sim \{x_1, x_2, \cdots x_s\}$, x_i is the Hensel code P-adic sequence with $x_i \sim \{a_{i0}, a_{i1}, \cdots, a_{in}; \text{point position}\}$ respect to prime $\{p_1, p_2, \cdots, p_s\}$.

The residue representation $r \sim \{r_1, r_2, \cdots, r_s\}$ can be given as:

$$r_i = p^{point\ position} \sum_{j=0}^{n} a_{ij} p^j$$

where $p \sim \{p_1^n, p_2^n, \cdots, p_s^n\}$.

For example, if we choose the prime set as {2147483647, 2147483629, 2147483587} (the largest prime numbers smaller than square root of 2 to 64 power, 64-bit CPU architecture, $p_i \leq 2147483647$), we wish to obtain the reflexive general inverse of matrix *A*, given in the following example. For each $GF(p)$ calculation, we choose the *P*-adic length as 2. The computation process is the following:

The entry rational matrix,

$$A = \begin{bmatrix} 1 & 2 \\ 1/3 & 1/4 \\ 5 & 6 \end{bmatrix}$$

After modulo operations by p_1, p_2, p_3, we have the following *P*-adic matrices,

$p_1 = 2147483647$,

$$\mathrm{A}_{\mathrm{P}_1} = \begin{bmatrix} .1, 0 & .2, 0 \\ .1431655765,\ 1431655764 & .536870912, 1610612735 \\ .5, 0 & .6, 0 \end{bmatrix}$$

$p_2 = 2147483629$

$$\mathrm{A}_{\mathrm{P}_2} = \begin{bmatrix} .1, 0 & .2, 0 \\ .1431655753, 1431655752 & .1610612722, 1610612721 \\ .5, 0 & .6, 0 \end{bmatrix}$$

$p_3 = 2147483587$

$$\mathrm{A}_{\mathrm{P}_3} = \begin{bmatrix} .1, 0 & .2, 0 \\ .1431655725, 1431655724 & .536870897, 1610612690 \\ .5, 0 & .6, 0 \end{bmatrix}$$

Parallel calculation of each p_i under *P*-adic arithmetic to get the reflexive general inverse, the results:

$p_1 = 2147483647$

$$g - Inverse(A)_{P_1} = \begin{bmatrix} .1717986917, 858993458 & .1288490193, 1717986917 & .0, 0 \\ .1288490189, 1717986917 & .429496727, 1288490188 & .0, 0 \end{bmatrix}$$

$p_2 = 2147483629$

$$g - Inverse(A)_{P_2} = \begin{bmatrix} .858993451, 1288490177 & .1717986908, 429496725 & .0, 0 \\ .1717986904, 429496725 & .1288490175, 858993451 & .0, 0 \end{bmatrix}$$

$p_3 = 2147483587$

$$g - Inverse(A)_{P_3} = \begin{bmatrix} .1717986869, 858993434 & .1288490157, 1717986869 & .0, 0 \\ .1288490153, 1717986869 & .429496715, 1288490152 & .0, 0 \end{bmatrix}$$

Decoding from the extended Chinese remainder theorem is the following:

$$g - Inverse(A) = \begin{bmatrix} -3/5 & 24/5 & 0 \\ 4/5 & -12/5 & 0 \end{bmatrix}.$$

4. Practical Considerations for the Implementation of Multiple P-adic Algorithm

4.1. *Advantages of multiple modulus arithmetic*

There are three advantages of multiple *P*-adic algorithm as stated below.

4.1.1. *Avoid the denominator problem*

For rational number $\frac{b}{a}$ and prime p, if a and p are not relatively prime, we cannot calculate $\frac{b}{a} \bmod p$. Because p is a prime, if a and p are not relatively prime, $a = xp^y, x, y \in N$. We can still get the finite *P*-adic sequence of $\frac{b}{a}$, just the point position will be equal to y.

4.1.2. *Increase the representation range*

With $\{p_1, p_2, \cdots, p_s\}$, $p = \prod_{i=1}^{s} p_i$, for multiple module arithmetic, the bound for the representation of denominator and/or numerator will be $\lambda\sqrt{p}$ ($\lambda = 0.618\cdots$ is a root of $\lambda^2 + \lambda - 1 = 0$). While for multiple *P*-adic algorithm with each *P*-adic length is r, the bound will be $\lambda\sqrt{p'}$, $p' = \prod_{i=1}^{s} p_i^r$.

4.1.3. *Parallel data structure*

One of the important issues of finite *P*-adic arithmetic is to choose the *P*-adic sequence length r. If the initial r is not long enough, Hensel code overflow will happen [6]. The *P*-adic sequence length r needs to be increased and the calculated results have to be discarded. On the other hand, for the multiple *P*-adic algorithm, when overflow happens, the calculated results can be kept. One should merely choose another prime p_i to continue the calculation, then combine the previously calculated results to convert back to the rational number by the extended Chinese remainder theorem.

By the "natural" structure of the extended Chinese remainder theorem, multiple *P*-adic arithmetic can be realized through parallel computation.

4.2. *Choosing a prime*

How to choose the prime set $\{p_1, p_2, \cdots, p_s\}$? According to the theory, for a fixed s value, the larger p_i you choose, the larger the bound that will result. But for computer architectures with 32 bit or 64 bit CPUs, when using the existing integer classes, the largest p_i should be chosen with respect to 46337 or 2147483647 to assure overflow protection [4]. This means that for a 32-bit CPU architecture, $p_i \leq 46337$, while for a 64-bit CPU architecture, $p_i \leq 2147483647$.

4.3. *Overflow detection [6]*

Extended CRT overflow: For the extended CRT systems with the prime set $\{p_1, \cdots p_s\}$,when a rational number $\frac{b}{a}$ mod p_i set $r \sim \{r_1, \cdots, r_s\}$, $GCD(a,b) = 1$, satisfies $|a,b| > \lambda\sqrt{p}, p = \prod_{i=1}^{s} p_i$ ($\lambda = 0.618\cdots$ is a root of $\lambda^2 + \lambda - 1 = 0$), the rational number, whose set cannot be uniquely recovered by the inverse transformation. We call this situation extended CRT overflow. One way to detect the overflow is to predict the bound, then decide the size of the prime set $\{p_1, p_2, \cdots p_n\}$ by Newman [12]. Another way to detect overflow is to provide some extra digits [6]. This method can detect the overflow by using the prime set $\{p_1, \cdots p_n\}$ and the residue number set itself. In this method, each number set should have some extra digits used for verification, the length is kept by *k*.
With prime set $\{p_1, p_2, \cdots p_i, p_{i+1}, \cdots, p_{i+k}\}$, for any rational number *x*, we get the set $\{r_1 = x \bmod p_1, \cdots, r_{i+k} = x \bmod p_{i+k}\}$, we record it as:

$$x \sim \{r_1, r_2, \cdots, r_i, r_{i+1}, \cdots, r_{i+k}\}$$

During the overflow detection process, it will be treated as

$$x \sim (r_0 r_1 \cdots r_i \quad \underbrace{r_{i+1} \cdots r_{i+k}}_{verification\ part} \ )$$

Notation: *Decoding(x, i)* and *Decoding(X, i)* will be used to donate decoding rational number set *x* and matrix *X* into rational number and rational number matrix by first *i* digits.

Overflow happened, if:

$$Decoding(x, i) \neq Decoding(x, i + k)$$

Overflow did not happen, if:

$$Decoding(x, i) = Decoding(x, i + k)$$

For example, taking prime set $\{11, 13, 17, 19\}$

$$x = \{1,6,10,17\}$$

$$y = \{1,11,0,6\}$$

Let the last one digit as verification part,

Decoding(x, 3) = 18/29

Decoding(x, 4) = 18/29

But,

Decoding(y, 3) = -34/37

Which is not equal to:

Decoding(y, 4) = -85/179

By the overflow detection method, *x=18/29* is correct, and for *y*, overflow happened.

This method has not been perfectly proved yet, but has a highly practical usage. If the prime set and verification part *k* is chosen properly, there will be no errors. With the values of prime set $\{p_1, p_2, \cdots p_i, p_{i+1}, \cdots, p_{i+k}\}$ increase, the possibility of error-happening decreases. For fixed prime set value, with the increase of verification part *k*, the possibility of error-happening also decreases. The experiments 1 to 3 will give support to this property.

Experiment 1:

Each time, the prime set $\{p_1, \cdots p_s\}$ is fixed, p_i is continuous series of prime numbers. For each prime set, we randomly generate 10,000 rational number $\frac{a}{b}(GCD(a,b) = 1), |a,b| \leq 10^{128}$. The size of prime set is 10, including verification part *k* = 1. When *Decoding(x, i)* = *Decoding(x, i+k)*, but *Decoding(x, i)*$\neq \frac{a}{b}$, it is record as one error.

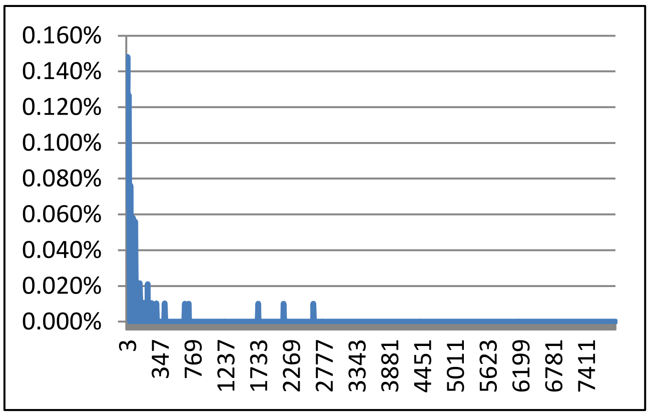


Fig. 3. Verification park *k* = 1, sequence length fixed, the error percentage
vertical axis: error percentage
horizontal axis: the value of p_1

From Fig. 3, we can see when $p_1 \geq 109$, the error percentage is extremely low. If $p_1 \geq 2777$, the error percentage goes to zero.

Experiment 2:

Each time, the prime set $\{p_1, \cdots p_s\}$ is fixed with the $p_1 = 3, p_i$ is continuous series of prime numbers. For each prime set, we randomly generate 10,000 rational numbers $\frac{a}{b}(GCD(a,b) = 1), |a,b| \leq 10^{128}$. The size of prime set is growing with 1 for each time, with verification part $k = 1$. When *Decoding(x, i) = Decoding(x, i+k)*, but *Decoding(x, i)*$\neq \frac{a}{b}$, it is record as one error.

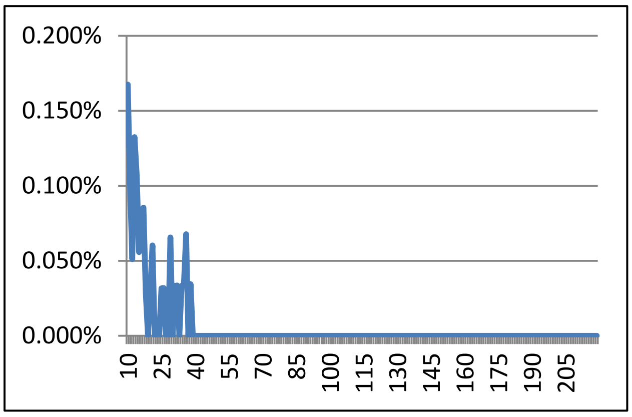


Fig. 4. Verification park $k = 1$, sequence length growing, the error percentage
vertical axis: error percentage
horizontal axis: the length of prime set

From Fig. 4, when the size of prime set is greater than 38, the error percentage goes to zero.

Experiment 3:

Each time, the prime set $\{p_1, \cdots p_s\}$ is fixed with the $p_1 = 3, p_i$ is continuous series of prime numbers. For each prime set, we randomly generate 10,000 rational numbers $\frac{a}{b}(GCD(a,b) = 1), |a,b| \leq 10^{128}$. The size of prime set is fixed with 30, and for each time, the verification part is growing with 1. When *Decoding(x, i) = Decoding(x, i+k)*, but *Decoding(x, i)*$\neq \frac{a}{b}$, it is record as one error.

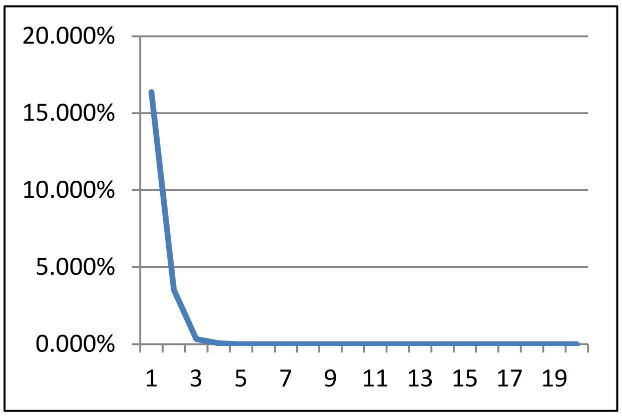


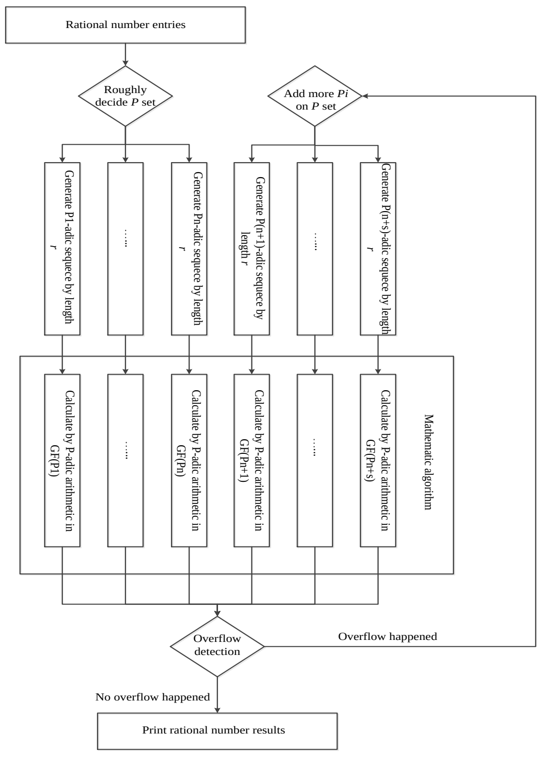
Fig. 5. Sequence length fixed with the verification part growing, the error percentage
vertical axis: error percentage
horizontal axis: the length of k

From Fig. 5, when the verification part is greater than 4, the error percentage goes to zero.

In the above experiments 2 and 3, we chose small primes to show the effects of prime set size and verification size. In practice, we chose the largest primes possible based on the CPU architecture. For 64-bit CPU architecture, we can choose primes close to 2147483647. Then the prime set size that we should select needs to be multiple of number of CPU cores, and the verification part should be as small as possible, usually just to be1.

4.4. *Parallel programming*

The modern computer architecture utilizes multiple cores in the CPU. The parallel tasking design can significantly improve the efficiency of any computation. The multiple *P*-adic arithmetic has the natural property to realize parallel computation. The programming design can be described by the flowing flow chart:



Rational number entries
Roughly decide *P* set
Add more *Pi* on *P* set
Generate P1-adic sequece by length *r*
……
Generate Pn-adic sequece by length *r*
Generate P(n+1)-adic sequece by length *r*
……
Generate P(n+s)-adic sequece by length *r*
Calculate by P-adic arithmetic in GF(P1)
……
Calculate by P-adic arithmetic in GF(Pn)
Calculate by P-adic arithmetic in GF(Pn+1)
……
Calculate by P-adic arithmetic in GF(Pn+s)
Mathematic algorithm
Overflow detection
Overflow happened
No overflow happened
Print rational number results

Fig. 6. *Multiple P-adic arithmetic implementation flow chart*

The number of tasks, which will be the same as s from $\{p_1, p_2, \cdots, p_s\}$, can be chosen with respect to the number of CPU cores to improve the efficiency.

5. Implementation of Multiple P-adic Arithmetic on Matrix

Our experiments were carried out on a typical laptop with Intel Core i5-2500 CPU as a parallel environment. The CPU has 4 cores for parallel processing.

5.1. *Moore-Penrose inverse [2]*

This algorithm is based on the Hermite theory [2], it is expressed as,

$$A^{+} = A^{t}(AA^{t}AA^{t})_{R}^{-}AA^{t}$$

A^{+} means the Moore-Penrose inverse of A (of order $m \times n$). M_R^- of $M = AA^tAA^t$ means the reflexive *g-inverse* of M.

Experiment 4:

We generated random matrices with size from 3 by 3 to 40 by 40, each element $\frac{a}{b}$ satisfies $|a, b| \leq 20$. For Multiple P-adic arithmetic algorithm, $s = 12$ for $p \sim \{p_1, p_2, \cdots, p_s\}$ and for each p the sequence length is 5. While for P-adic arithmetic, the sequence is 60. For each matrix size, we generated 30 simples. Both algorithms are used to calculate the Moore-Penrose inverse.

We use NTL [15] to represent larger integers for the

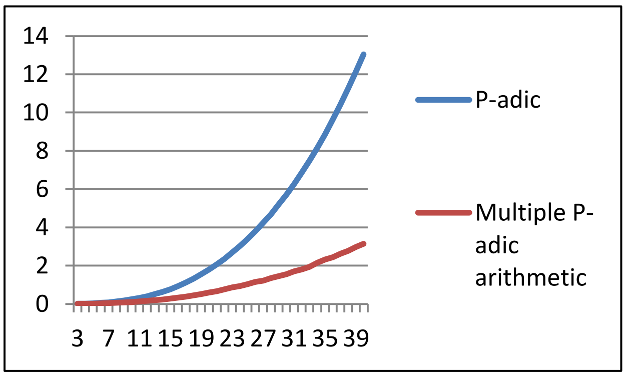


Fig. 7. *Moore-Penrose Inverse*
vertical axis: the average implementation time in second
horizontal axis: the matrix size

experiments. The speed up is defined as:

$$\text{Speed-up Rate} = \frac{P\text{−}adic}{Multiple\ P\text{−}adic}$$

5.2. *Polynomial method to calculate e^{At}[7, 13]*

For the calculation of e^{At}, the polynomial method is: Transform a $n \times n$ matrix A into Lower Hessenberg form *H*, and get the transforming matrix *T*,

$$T^{-1}AT = H$$

Convert the lower Hessenberg matrix H to Frobenius form according to the formula of Wilkinson [3],

$$C^{-1}WC = F$$

Form a diagonal matrix *D* that is supposed to transform the matrix so that the sub-diagonal consists of 1s,

$$D^{-1}FD = G$$

After the three transforming steps, we get the Frobenius canonical form *G*, invertible matrix *W* and its inverse matrix W^{-1}, for which $W^{-1} = D^{-1}C^{-1}T^{-1}$, *W=TCD*. Mostly, *G* will have the structure as the following:

$$G = \begin{bmatrix} 0 & & & & c_0 \\ 1 & 0 & & & c_1 \\ & \ddots & \ddots & & \vdots \\ & & 1 & 0 & c_{n-2} \\ & & & 1 & c_{n-1} \end{bmatrix}$$

According to the Cayley-Hamilton theorem, $A^n = c_0 I + c_1 A + \cdots + c_{n-1}A^{n-1}$

And it follows that any power of A can be expressed in terms of $I, A, \cdots A^{n-1}$:

$$A^k = \sum_{j=0}^{n-1} \beta_{kj} A^j$$

Then e^{At}can be implied as the following:

$$e^{tA} = \sum_{k=0}^{\infty} \frac{t^k A^k}{k!} = \sum_{k=0}^{\infty} \frac{t^k}{k!} \left[\sum_{j=0}^{n-1} \beta_{kj} A^j \right]$$
$$= \sum_{j=0}^{n-1} \left[\sum_{k=0}^{\infty} \beta_{kj} \frac{t^k}{k!} \right] A^j = \sum_{j=0}^{n-1} \alpha_j(t) A^j$$

where$\alpha_j(t)$ and β_{kj} are expressed as,

$$\alpha_j(t) = \sum \beta_{kj} t^k / k!$$

$$\beta_{kj} = \begin{cases} \delta_{kj} & (k < n) \\ c_j & (k = n) \\ c_0\beta_{k-1,n-1} & (k > n, j = 0) \\ c_j\beta_{k-1,n-1} + \beta_{k-1,j-1} & (k > n, j > 0) \end{cases}$$

Experiment 5:

We generated random matrices with size from 3 by 3 to 40 by 40, each element $\frac{a}{b}$ satisfies $|a, b| \leq 20$. For Multiple P-adic arithmetic algorithm, $s = 12$ for $p \sim \{p_1, p_2, \cdots, p_s\}$ and for each p the sequence length is 5. While for P-adic arithmetic, the sequence is 60. For each matrix size, we generated 30 simples. Both algorithms are used to calculate $e^{At}, t = 1$ with 100 iterations.

From the above two experiments, we can find that on the 4 cores CPU (Intel Core i5- 2500), the multiple P-adic arithmetic algorithm will speed up about 2 to 4 times based on the matrix sizes compared with that of direct P-adic arithmetic.

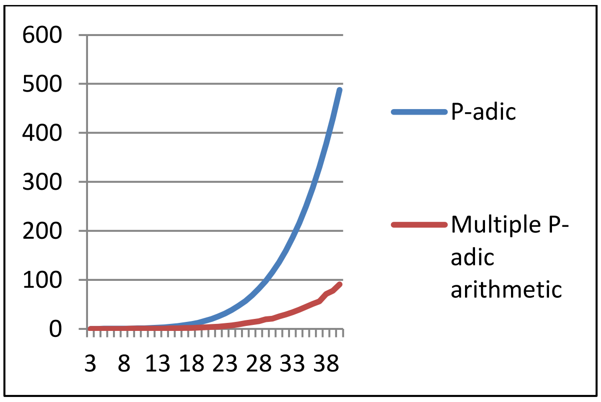


Fig. 8. *Polynomial method to calculate* e^{At}
vertical axis: the average implementation time in second
horizontal axis: the matrix size

6. Efficiency Analysis and Conclusions

Experiment 6:

We generated random matrices with size from 3 by 3 to 40 by 40, each element $\frac{a}{b}$ satisfies $|a, b| \leq 20$. For the multiple P-adic arithmetic algorithm, $s \sim \{4, 5, 8, 12\}$ for $p \sim \{p_1, p_2, \cdots, p_s\}$ and for each p the sequence length is 5. While for P-adic arithmetic, the sequence is $\{20, 25, 40, 60\}$. For each matrix size, we generated 30 simples. Both algorithms are used to calculate the Moore-Penrose inverse.

We get the average of speed up rate ($\frac{P-adic}{Multiple\ P-adic}$) for each size s as shown in Figures 9 and 10.

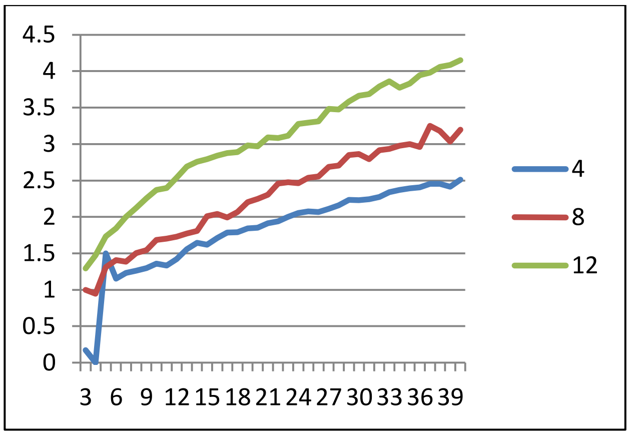


Fig. 9. Speed up rate for s equal to 4, 8 and 12
vertical axis: speed up rate value
horizontal axis: the matrix size

From Fig. 9, we can see that with the increase of the integer sequence length for multiple P-adic and P-adic sequences, we will have more advantage of the multiple P-adic arithmetic. The reason is that as the length increase, the time complexity for P-adic arithmetic is $O(n^2)$, while for Multiple P-adic arithmetic is $O(n)$.

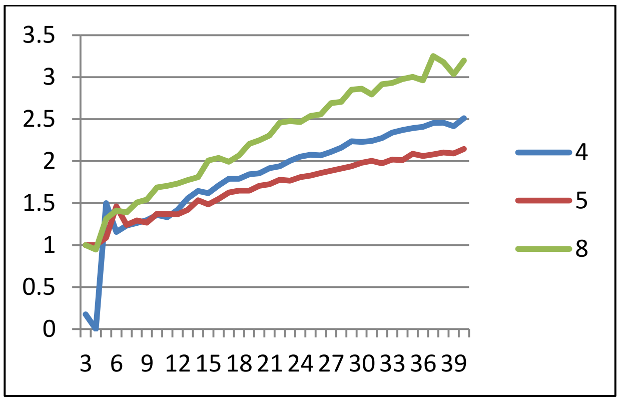


Fig. 10. *Speed up rate for s equal to 4, 5 and 8*
vertical axis: speed up rate value
horizontal axis: the matrix size

The CPU architecture can be an important part of the speeding up. From Fig. 9 and Fig. 10, we can see that if the length is a multiple of the number of CPU cores, the speed up is outstanding; while when the length is not a divisible number by CPU cores, such as 5 for a CPU with four cores, the speed-up will be poor. Also, as the matrix sizes grow, the speed-up factor becomes even more significant.

Acknowledgements

This research project is supported by the Air Force Office of Scientific Research, FA9550-11-1-0315.

References

1. E. V. Krishnamurthy, "Matrix Processors Using *P*-adic Arithmetic for Exact Linear Computations", Vol. C-26, No. 7, July 1977.
2. T. M. Rao, K. Subramanian and E.V. Krishnamurthy, "Residue Arithmetic Algorithms for Exact Gomputation of g-inverses of Matrices", SIAM J. NUMER. ANAL, Vol. 13, No. 2, pp. 155-171, April 1976.
3. J. D. Dixon, "Exact Solution of Linear Equations Using P-adic Expansions", Number. Math. 40, 137-141(1982), Springer- Verlag.
4. X. Li, M, Zhao and C. Lu, "Efficient Algorithms and Implementation for Error-free Computation Using *P*-adic", CSNI2011.
5. C. Lu, X. Li and L. Shan, "Periodicity of the *P*-adic Expansion after Arithmetic Operations in *P*-adic Field", ACIS2012.
6. X. Li, C. Lu and J. A. Sjogren, "A Method for Hensel Code Overflow Detection", ACM SIGAPP Applied Computing Review, Vol. 12, Issue 1, p. 6-11, 2012.
7. X. Li, M. Zhao, C. Lu and J. A. Sjogren, "Implementation of the Polynomial Method to Calculate e^{At} Using *P*-adic", Proceedings of the 2012 ACM Research in Applied Computation Symposium, 2012.
8. P. Kornerup and D. W. Matula, Finite Precision Number Systems and Arithmetic, Cambridge University Press, 2010.
9. J. Morrison, "Parallel *P*-adic computation, Information Processing Letters", Vol. 28, Issue 3, 1988.
10. C. Limongelli and H. W. Loidl, "Rational Number Arithmetic by Parallel *P*-adic Algorithms", Springer Verlag, editor, Proc. Of Second International Conference of the Austrian Center for Parallel Computation (ACPC), Vol. 734 of LNCS, 1993.
11. C. K. Koc, "Parallel *P*-adic Method for Solving Linear Systems of Equations", Parallel Computing, Vol. 23(13), 1997.
12. M. Newman, "Solving Equations Exactly", Mathematics and Mathematical Physics, Vol. 71B, No. 4, Oct-Dec 1967.
13. C. Moler and C. V. Loan, "Nineteen Dubious Ways to Compute the Exponential of Matrix", Twenty-Five Years Later, Society for Industrial and Applied Mathematics, Vol. 45, No. 1, 2003.
14. C. Limongelli and R. Pirastu, "*p*-adic Arithmetic and Parallel Symbolic Computation: An Implementation for Solving Linear Systems", Technical Report N. RT-INF-1-1995.
15. V. Shoup, NTL library at: http://www.shoup.net/ntl/.