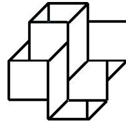


Laboratório Nacional de Computação Científica

Programa de Pós Graduação em Modelagem Computacional

Paralelização Eficiente para o Algoritmo de Exponenciação Modular

Por

Pedro Carlos da Silva Lara

PETRÓPOLIS, RJ - BRASIL

MARÇO DE 2011

PARALELIZAÇÃO EFICIENTE PARA O ALGORITMO DE EXPONENCIAÇÃO MODULAR

Pedro Carlos da Silva Lara

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO LABORATÓRIO NACIONAL DE COMPUTAÇÃO CIENTÍFICA COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM MODELAGEM COMPUTACIONAL

Aprovada por:

____

Prof. Renato Portugal, D.Sc

(Presidente)

____

Prof. Eduardo Lúcio Mendes Garcia, D.Sc.

____

Prof. Nadia Nedjah, Ph.D

PETRÓPOLIS, RJ - BRASIL
MARÇO DE 2011

Lara, Pedro Carlos da Silva

L318p paralelização eficiente para o algoritmo de exponenciação modular / Pedro Carlos da Silva Lara. Petropólis, RJ. : Laboratório Nacional de Computação Científica, 2011.

LX, 60 p. : il.; 29 cm

Orientador: Renato Portugal

Dissertação (M.Sc.) – Laboratório Nacional de Computação Científica, 2011.

1. Criptografia 2. Exponenciação Modular 3. Algoritmos Paralelos 4. Balanceamento de Carga I. Portugal, Renato. II. LNCC/MCT. III. Título.

CDD – 652.8

Feliz aquele que transfere o que sabe e

aprende o que ensina.

Cora Coralina

Dedicatória

A minha mãe, pelo apoio incondicional e
pelo incentivo a busca de novos
conhecimentos.

Agradecimentos

Agradeço ao meu orientador professor Renato Portugal e aos professores Fábio Borges e Nadia Nedjah pelo apoio crucial no desenvolvimento e elaboração desta dissertação.

Resumo da Dissertação apresentada ao LNCC/MCT como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

PARALELIZAÇÃO EFICIENTE PARA O ALGORITMO DE EXPONENCIAÇÃO MODULAR

Pedro Carlos da Silva Lara

MARÇO , 2011

Orientador: Renato Portugal, D.Sc

Algoritmos de exponenciação modular tem sido utilizados de maneira central em grande parte da criptografia assimétrica, geração de números aleatórios e testes de primalidade. Este trabalho propõe novas técnicas de paralelização para o algoritmo de exponenciação modular que incluem métodos de paralelização massiva e balanceamento de carga. São avaliados resultados teóricos e práticos acerca dos métodos propostos.

Abstract of Dissertation presented to LNCC/MCT as a partial fulfillment of the requirements for the degree of Master of Sciences (M.Sc.)

FAST PARALLEL METHODS FOR MODULAR EXPONENTIATION

Pedro Carlos da Silva Lara

November, 2011

Advisor: Renato Portugal, D.Sc

Modular exponentiation algorithms play an important role in many asymmetric cryptography, random number generation and primality tests. This work proposes new techniques for parallelizing the algorithm of modular exponentiation methods both in terms of massive parallelism and load balancing techniques. The theoretical and practical results of the methods are assessed in this work.

Sumário

1 Introdução 2

2 Aritmética Inteira de Pecisão Múltipla 4

2.1 Algoritmo para a Adição e Subtração 4

2.2 Multiplicação 6

2.2.1 Karatsuba-Ofman 7

2.3 Cálculo de Quadrados 8

2.4 Divisão 8

2.5 Redução Modular 10

2.5.1 Redução Modular de Montgomery 10

2.5.2 Redução Modular de Barrett 12

2.6 Conclusões 12

3 Criptografia Assimétrica 13

3.1 Problema do Logaritmo Discreto 13

3.2 Problema da Fatoração de Inteiros 14

3.3 Algoritmo de Diffie-Helman 14

3.3.1 Ataque Ativo Man-in-the-Middle 15

3.4 RSA 16

3.5 ElGamal 19

3.6 Conclusões 21

4 Paralelização para a Exponenciação Modular 22

4.1 Algoritmo Binário para Exponenciação Modular 22

4.2 Paralelização do Algoritmo Binário para um Número Pequeno de Processadores 24

4.2.1 Análise de Complexidade 26

4.3 Paralelização Massiva 29

4.3.1 Análise de Complexidade 29

4.4 Resultados 30

4.5 Conclusões 33

5 Balanceamento de Carga 35

5.1 Balanceamento 35

5.2 Análise Experimental 41

5.3 Conclusão 42

Referências Bibliográficas 43

Apêndice

A Teste de Primalidade Miller-Rabin 47

Lista de Figuras

Figura

3.1 Ataque *man-in-the-middle* ativo. 16

4.1 Tempo de execução (a) e *speedup* (b) para até 48 processadores usando 256 bits 31

4.2 Tempo de execução (a) e *speedup* (b) para até 48 processadores usando 512 bits 31

4.3 Tempo de execução (a) e *speedup* (b) para até 48 processadores usando 1024 bits 31

4.4 Desempenho experimental para o algoritmo 21. (a) usando 512 bits (b) 1024 bits 32

4.5 Comparativo entre os algoritmo de paralelização apresentados. . . . 33

5.1 Número ótimo de processadores k em função do número de bits n do expoente. 39

5.2 Comparação entre os três algoritmos de paralelização apresentados. 41

Lista de Tabelas

Tabela

3.1 Complexidades dos algoritmos de fatoração de inteiros 14

3.2 Os números RSA que foram fatorados. 20

Lista de Algoritmos

1 Adição entre inteiros de tamanho arbitrário. 5

2 Subtração entre inteiros de tamanho arbitrário. 5

3 Multiplicação (*scanning form*). 6

4 Multiplicação de Karatsuba-Ofman 8

5 Cálculo de quadrado. 9

6 Divisão com resto entre inteiros de tamanho arbitrário. 9

7 Multiplicação Modular Clássica. 10

8 Redução modular de Montgomery. 11

9 Multiplicação modular de Montgomery. 11

10 Redução modular de Barrett. 12

11 Geração de chaves RSA. 17

12 Criptografia RSA. 17

13 Decifragem RSA. 18

14 Geração de chaves ElGamal. 20

15 Cifragem ElGamal. 20

16 Decifragem ElGamal. 20

17 Algoritmo binário versão esquerda para direita. 23

18 Algoritmo binário versão direita para esquerda. 23

19 Paralelização do algoritmo binário. 25

20 Paralelização com n linhas de execução. 27

21 Paralelização da linha 21 do algoritmo 20. 29

22 Teste probabilístico de primalidade Miller-Rabin. 48

Capítulo 1

Introdução

Whitfield Diffie e Martin E. Hellman Diffie e Hellman (1976) propuseram uma interessante solução para o problema de estabelecer uma chave secreta para dois usuários em um canal de comunicação inseguro, que é considerada a primeira prática de criptografia de chave pública. Seja p um primo e g um gerador do grupo cíclico $\mathbb{Z}_p^*$. Vamos supor que Alice e Bob queiram combinar uma chave secreta. Inicialmente, Alice escolhe aleatoriamente um inteiro secreto a tal que $a \in \mathbb{Z}_p$ e envia para Bob $k_a = g^a \mod p$. Analogamente, Bob seleciona um inteiro secreto $b \in \mathbb{Z}_p$ e envia à Alice $k_b = g^b \mod p$. Agora ambos usam seu inteiro secreto para obter uma chave secreta compartilhada $k_{ab} = (k_a)^b = (k_b)^a = g^{ab} \mod p$. É fácil perceber a importância de um algoritmo de exponenciação modular rápido no protocolo Diffie-Hellman. O RSA (Ron **R**ivest, Adi **S**hamir e Len **A**dleman) foi publicado em 1978 Rivest et al. (1978) e é atualmente o principal criptossistema de chave pública usado em aplicações comerciais. A segurança deste método está baseada na ausência de um algoritmo eficiente para o Problema da Fatoração de Inteiros (PFI). O criptossistema RSA consiste em três partes - geração de chaves, encriptação e decriptação. As duas últimas utilizam diretamente a exponenciação modular. Logo, o desempenho deste criptossistema assimétrico está estreitamente ligada com o desempenho da exponenciação modular Nedjah e Mourelle (2002). A implementação em *hardware* do RSA é discutida em Brickell (1990).

Brickell, Gordon, McCurley and Wilson (BGMW) em Brickell et al. (1992)

usaram a pré-computação de algumas potências para acelerar a exponenciação. Em Brickell et al. (1995) os mesmos autores propuseram duas paralelizações usando pré-computação. Outras paralelizações relacionadas ao BGMW são discutidas em Lim e Lee (1994). Deste modo, estas técnicas são particularmente interessantes em métodos que utilizam base fixa, por exemplo, protocolo Diffie-Hellman. O uso do RSA com múltiplos primos RSA Labs (2000) (*multi-prima RSA*) juntamente com o Teorema do Resto Chinês fornece uma paralelização direta e relativamente eficiente. N. Nedjah e L. M. Mourelle em Nedjah e Mourelle (2007) discutem e comparam a paralelização de alguns dos principais algoritmos de exponenciação modular.

Este trabalho propõe paralelizações para o algoritmo de exponenciação modular. São aboradas técnicas de balanceamento de carga que modifica e distribui melhor a carga entre os processadores para o algoritmo proposto em Lara et al. (2009). São mostrados os resultados teóricos e práticos que envolvem o balanceamento de carga proposto. O balanceamento de carga entre os processadores permitiu a redução significativa do número de processadores ótimo e também a redução do tempo de execução final para o cálculo da exponenciação modular.

O trabalho foi organizado da seguinte maneira: no Capítulo 2 é feita uma revisão sobre os principais algoritmos usados na aritmética de precisão múltipla. no Capítulo 3 será mostrado alguns dos principais algoritmos de criptografia assimétrica, que concentra grande parte da motivação para este trabalho. No Capítulo 4, são propostas paralelizações iniciais para o algoritmo de exponenciação modular. São mostrados resultados práticos e teoricos acerca do método proposto. No Capítulo 5 são apresentadas melhorias na paralelização introduzida no Capítulo 4. Neste caso, foram exploradas técnicas de balanceamento de cargas para acelerar o tempo computacional e diminuir o número de processadores requeridos.

Capítulo 2

Aritmética Inteira de Pecisão Múltipla

Grande parte dos esquemas criptográficos assimétricos necessitam de aritmética no anel $\mathbb{Z}_m$. Atualmente, o algoritmo assimétrico RSA Rivest et al. (1978) utiliza, para uma segurança aceitável, $\log_2 m \approx 1024$. Ou seja, todos os cálculos são feitos com números inteiros de aproximadamente 1024 *bits*. Neste Capítulo é feito uma breve revisão dos conceitos que norteiam a implementação prática da aritmética de inteiros de tamanho arbitrário. As idéias deste Capítulo foram inspiradas nas referências Knuth (1997) e Menezes et al. (1997). Para a implementação em *software* uma importante referência é Denis (2006).

2.1 Algoritmo para a Adição e Subtração

Para a adição e subtração serão consideradas entradas cujas representações em uma determinada base b sejam do mesmo tamanho. Caso suas representações na base b sejam de diferentes comprimentos, a menor entrada será completada com 0's a esquerda até que tenham o mesmo tamanho. Por exemplo, dadas as entradas 15067 e 594 e a base $b = 16$ desta forma será considerado $15067 = (3ADB)_{16}$ e $594 = (0252)_{16}$. O algoritmo 1 descreve a computação de uma adição e o algoritmo 2 a subtração entre dois inteiros de tamanhos arbitrários.

Para a implementação prática, a base b está relacionada ao tamanho da palavra do processador utilizado (atualmente é comum o uso de processadores 32 e 64 bits, assim, b poderia ser 32 ou 64 respectivamente). Também poderá ser

Algoritmo 1: Adição entre inteiros de tamanho arbitrário.

Entrada: Inteiros positivos $x = (x_n \dots x_0)_b$ e $y = (y_n \dots y_0)_b$.

Saída: $x + y = (r_{n+1} r_n \dots r_0)$.

```
1  início
2  |  carry ← 0;
3  |  para i ← 0 até n faça
4  |  |  r_i ← x_i + y_i + carry  mod b;
5  |  |  se (x_i + y_i + carry) > b então
6  |  |  |  carry ← 1;
7  |  |  senão
8  |  |  |  carry ← 0;
9  |  r_{n+1} = carry;
10 |  retorna (r_{n+1} r_n ... r_0)_b;
11 fim
```

Algoritmo 2: Subtração entre inteiros de tamanho arbitrário.

Entrada: Inteiros positivos $x = (x_n \dots x_0)_b$ e $y = (y_n \dots y_0)_b$, com $x \geq y$.

Saída: $x - y = (r_n r_{n-1} \dots r_0)$.

```
1  início
2  |  carry ← 0;
3  |  para i ← 0 até n faça
4  |  |  r_i ← x_i − y_i + carry  mod b;
5  |  |  se (x_i − y_i + carry) < 0 então
6  |  |  |  carry ← −1;
7  |  |  senão
8  |  |  |  carry ← 0;
9  |  retorna (r_n r_{n−1} ... r_0)_b;
10 fim
```

utilizada a instrução, em linguagem assembly, JC (*Jump if Carry*), que captura a ocorrência de *carry*, ou seja "vai um" no bit mais significativo da instrução anterior. A maioria dos processadores modernos permite este mecanismo.

2.2 Multiplicação

Nesta seção serão apresentados alguns dos principais algoritmos para a multiplicação entre inteiros de tamanho arbitrário. Se o inteiro x possui $n+1$ dígitos na base b e y possui $m+1$ também na base b então $x \cdot y$ possui, no máximo, $m+n+2$ dígitos na base b. Nos algoritmos que seguem, o símbolo $(ts)_b$ significa a concatenação de dois inteiros $t, s \in \{0, 2^b - 1\}$. O algoritmo 3 é conhecido como *scanning form*

Algoritmo 3: Multiplicação (*scanning form*).

```
   Entrada: Inteiros positivos x = (x_n ... x_0)_b e y = (y_m ... y_0)_b.
   Saída: x · y = (r_{n+m+1} r_{n+m} ... r_0).
 1 início
 2 │ r_i ← 0, i ∈ {0, n + m + 1};
 3 │ para i ← 0 até m faça
 4 │ │ k ← 0;
 5 │ │ para j ← 0 até n faça
 6 │ │ │ (ts)_b ← x_j · y_i + r_{i+j};
 7 │ │ │ k ← t;
 8 │ │ └ r_{i+j} ← s;
 9 │ └ r_{i+m+1} ← t;
10 │ retorna (r_{n+m+1} r_{n+m} ... r_0)_b;
11 fim
```

O algoritmo calcula o produto $x \cdot y$ usando $T(x, y) = (n+1)(m+1)$ multiplicações na base b. Para implementações práticas usa-se a base b sendo a metade do tamanho a palavra do processador utilizado. Assim o inteiro $(ts)_b$ possui, no máximo, o tamanho da palavra do processador, neste caso $2 \cdot b$. Portanto, a linha 6 do algoritmo 3 poderá ser processada diretamente, sem usar outros mecanismos. Outra alternativa seria utilizar inteiros de precisão dupla para a receber o resultado da linha 6 do algoritmo 3.

2.2.1 Karatsuba-Ofman

O algoritmo mostrado na seção anterior possui complexidade computacional $O(n^2)$ para entradas de n bits. O algoritmo proposto por Anatoly A. Karatsuba and Y. Ofman Karatsuba e Ofman (1963) utiliza o conceito de divisão e conquista para o cálculo da multiplicação entre números de tamanho arbitrário Knuth (1997). A complexidade deste algoritmo é $O(n^{1.58})$. Considere dois inteiros, x, y de n *bits* cada um. Seja $k = \lceil \frac{n}{2} \rceil$ temos a seguinte decomposição de x e y

$$x = x_H 2^k + x_L$$

$$y = y_H 2^k + y_L$$

sendo que x_L, x_H e y_L, y_H possuem k bits em sua representação binária. Neste caso, podemos calcular o produto $x \cdot y$ da seguinte forma:

$$\begin{aligned} x \cdot y = & \ (x_H 2^k + x_L)(y_H 2^k + y_L) = x_H y_H 2^{2k} + (x_H y_L + y_H x_L) 2^k + x_L y_L \\ = & \ x_H y_H 2^{2k} + [(x_H + x_L)(y_H + y_L) - x_H y_H - x_L y_L] 2^k + x_L y_L \end{aligned} \quad (2.1)$$

Da equação 2.1 temos que o cálculo de $x \cdot y$ poderá ser computado com 3 multiplicações de $k = \frac{n}{2}$ bits. O processo poderá prosseguir recursivamente da forma que foi mostrado na equação 2.1 até que tenhamos o inteiro k sendo menor ou igual ao tamanho da palavra do processador W. Estas idéias estão sistematizadas no algoritmo 4.

O custo computacional $T(n)$ para o algoritmo 4 pode ser deduzido da seguinte forma:

$$T(n) = \overbrace{2T(\frac{n}{2})}^{\text{linha 6 e 7}} + \underbrace{T(\frac{n}{2} + 1)}_{\text{linha 8}} + \overbrace{\delta_0}^{\text{linha 9}} = 3T(\frac{n}{2}) + \delta_0' \quad (2.2)$$

Da equação 2.2 temos que o valor δ_0' poderá ser desconsiderado uma vez que que a operação de soma e deslocamento binário à esquerda são substancialmente mais rápidas que a operação de multiplicação. Tomando $T(1) = 1$ temos a seguinte

Algoritmo 4: Multiplicação de Karatsuba-Ofman

Entrada: Inteiros positivos x e y com n bits.
Saída: $x \cdot y$.

```
1  início
2  |  k ← n;
3  |  se k < W então
4  |  |  retorna x · y;
5  |  senão
6  |  |  P1 ← Karatsuba-Ofman(x_H, y_H);
7  |  |  P2 ← Karatsuba-Ofman(x_L, y_L);
8  |  |  P3 ← Karatsuba-Ofman(x_H + y_L, y_H + x_L);
9  |  |  retorna (P1 ≪ k) + [(P3 − P2 − P1) ≪ k/2] + P2;
10 fim
```

relação

$$\begin{aligned} T(n) &\approx 3T(\tfrac{n}{2}) \\ T(n) &\approx 3 \cdot 3T(\tfrac{n}{4}) \\ T(n) &\approx 3 \cdot 3 \cdot 3T(\tfrac{n}{8}) \\ &\vdots \\ T(n) &\approx \overbrace{3 \cdot 3 \cdot \ldots \cdot 3}^{\log_2 n \text{ vezes}} = 3^{\log_2 n} = n^{\log_2 3} = n^{1.58}. \end{aligned}$$

2.3 Cálculo de Quadrados

Em muitos casos é importante que o cálculo de x^2 seja bastante eficaz. O algoritmo 5 calcula o quadrado de um inteiro de tamanho arbitrário. Este algoritmo está baseado no algoritmo 3, no entanto, possui importantes modificações.

O número de multiplicações simples para o algoritmo 5 fica determinado por $T(n) = \frac{(n+1)^2}{2}$. A multiplicação por 2, na linha 8, é ignorada na expressão $T(n)$ pois esta operação é executada, na prática, usando um deslocamento à esquerda.

2.4 Divisão

O algoritmo , da divisão, é a operação mais custosa entre as operações básicas de inteiros com precisção múltipla (operações mostradas anteriormente). O algoritmo 6 ilustra o funcionamento do algoritmo de divisão inteira (com resto).

Algoritmo 5: Cálculo de quadrado.

```
   Entrada: Inteiros positivos x = (x_n ... x_0)_b
   Saída: x · x = x^2 = (r_{2n+1} r_{2n} ... r_0).
 1 início
 2     r_i ← 0, i ∈ {0, 2n+1};
 3     para i ← 0 até n faça
 4         (ts)_b ← x_i · x_i + r_{2i};
 5         r_{2i} ← s;
 6         k ← t;
 7         para j ← i+1 até n faça
 8             (ts)_b ← 2x_j · x_i + r_{i+j} + k;
 9             k ← t;
10             r_{i+j} ← s;
11         r_{i+n} ← t;
12     retorna (r_{2n+1} r_{2n} ... r_0)_b;
13 fim
```

Algoritmo 6: Divisão com resto entre inteiros de tamanho arbitrário.

```
   Entrada: Inteiros positivos x = (x_n ... x_0)_b e y = (y_m ... y_0)_b, com n ≥ m e
            y ≠ 0.
   Saída: O quociente q = (q_{n-m} q_{n-m-1} ... q_0)_b e o resto r = (r_m r_{m-1} ... r_0)_b
          onde x = qy + r e 0 ≤ r < y.
 1 início
 2     q_i ← 0, i ∈ {0, n-m};
 3     enquanto x ≥ yb^{n-m} faça
 4         q_{n-m} ← q_{n-m} + 1;
 5         x ← x - yb^{n-m};
 6     para i = n até m+1 faça
 7         se x_i = y_m então
 8             q_{i-m-1} ← b - 1;
 9         senão
10             q_{i-m-1} ← ⌊(x_{i-1} + x_i b)/y_m⌋;
11         enquanto q_{i-m-1}(y_{m-1} + y_m b) > x_{i-2} + x_{i-1} b + x_i b^2 faça
12             q_{i-m-1} ← q_{i-m-1} - 1;
13         x ← x - q_{i-m-1} y b^{i-m-1};
14         se x < 0 então
15             x ← x + y b^{i-m-1};
16             q_{i-m-1} ← q_{i-m-1} - 1;
17     r ← x;
18     retorna q, r;
19 fim
```

2.5 Redução Modular

Grande parte da criptografica assimétrica é baseada na aritmética módulo m, ou seja, no anel $\mathbb{Z}_m$ (*e.g.* RSA, ElGamal, Diffie-Helman). Nesta seção serão explanados algoritmos usados para o cálculo de $a \mod m$ onde $a > m$.

2.5.1 Redução Modular de Montgomery

Com o que já foi mostrado nas seções anteriores, já poderíamos definir um algoritmo para o cálculo de $x \mod m$ (veja algoritmo 6). No entanto, o algoritmo de redução modular 6 é ineficiente no caso geral. Uma computação muito comum em criptografia é a multiplicação modular. O algoritmo 7 calcula o produto de dois inteiros x, y módulo m.

Algoritmo 7: Multiplicação Modular Clássica.

Entrada: Inteiros positivos x, y e m.
Saída: $r = x \cdot y \mod m$.

1 **início**
2     $r \leftarrow x \cdot y$ (algoritmo 4);
3     $r \leftarrow r \mod m$ (algoritmo 6);
4   **retorna** r;
5 **fim**

Montgomery Montgomery (1985) propôs um algoritmo que calcula eficientemente a redução modular. Sejam R e T dois inteiros positivos tais que $R > m$ e $mdc(R, m) = 1$ e $0 \leq T < mR$. A técnica que será mostrada nesta Seção calcula $TR^{-1} \mod m$ sem utilizar os passos do algoritmo 7. Isto é denominado *redução de T módulo m usando R* (veja o Algoritmo 8). Dependendo da escolha de R, esta redução pode ser eficientemente calculada. Podemos estender este algoritmo para o cálculo da multiplicação modular (Algoritmo 9). Para uma única multiplicação modular o algoritmo de Montgomery não é eficiente, uma vez que este algoritmo fornece $xyR^{-1} \mod m$. Assim o custo para transformar a saída do algoritmo 9 em $xy \mod m$ não justifica o seu uso. No entanto, na exponenciação este algoritmo é bastante interessante.

Algoritmo 8: Redução modular de Montgomery.

Entrada: $m = (m_{k-1} \dots m_0)_b$ com $mdc(m, b) = 1$, $R = b^n$, $m' = -m^{-1} \mod b$ e $T = (t_{2k-1} \dots t_0)_b$

Saída: $TR^{-1} \mod m$.

```
1  início
2     W ← T (W é da forma W = (w_{2k-1} ... w_0)_b);
3     para i = 0 até k − 1 faça
4        u_i ← w_i m' mod b;
5        W ← W + u_i m b^i;
6     W ← W/b^n;
7     se W ≥ m então
8        W ← W − m;
9     retorna W;
10 fim
```

Algoritmo 9: Multiplicação modular de Montgomery.

Entrada: $m = (m_{k-1} \dots m_0)_b$ com $mdc(m, b) = 1$, $R = b^n$, $m' = -m^{-1} \mod b$ e $x = (x_{k-1} \dots x_0)_b$, $y = (y_{k-1} \dots y_0)_b$ com $0 < x, y < m$

Saída: $xyR^{-1} \mod m$.

```
1  início
2     W ← 0 (W é da forma W = (w_k ... w_0)_b);
3     para i = 0 até k − 1 faça
4        u_i ← (w_0 + x_i y_0) m' mod b;
5        W ← (W + x_i y + u_i m)/b;
6     se W ≥ m então
7        W ← W − m;
8     retorna W;
9  fim
```

2.5.2 Redução Modular de Barrett

O algoritmo de Barrett Barrett (1987), descrito em 10, calcula a redução modular $y = x \mod m$ (com $x > m$) usando a pré-computação do termo $\mu = \lfloor b^{2k}/m \rfloor$, onde b é a base da representação dos números x e m (por exemplo, $b = 2$, representação binária) k é o número de dígitos de m na base b. No Algoritmo 10, x deverá ter no máximo $2k$ dígitos na base b. Este algoritmo é especialmente interessante quando são necessárias muitas reduções modulares usando um único m (por exemplo, no RSA). Assim o termo μ poderá ser computado uma única vez.

Algoritmo 10: Redução modular de Barrett.

Entrada: Inteiros positivos $x = (x_{2k-1} \dots x_0)_b$ e $m = (m_{k-1} \dots m_0)_b$ e $\mu = \lfloor b^{2k}/m \rfloor$.
Saída: $y = x \mod m$.

1 **início**
2 $q_1 \leftarrow \lfloor x/b^{k-1} \rfloor$;
3 $q_2 \leftarrow q_1 \cdot \mu$;
4 $q_3 \leftarrow \lfloor q_2/b^{k+1} \rfloor$;
5 $y_1 \leftarrow x \mod b^{k+1}$;
6 $y_2 \leftarrow q_3 \cdot m \mod b^{k+1}$;
7 $y \leftarrow y_1 - y_2$;
8 **se** $y < 0$ **então**
9 $y \leftarrow y + b^{k+1}$;
10 **enquanto** $r \geq m$ **faça**
11 $y \leftarrow y - m$;
12 **retorna** y;
13 **fim**

2.6 Conclusões

Este Capítulo revisou os principais algoritmos para aritmética de precisão múltipla. Todos os algoritmos mostrados neste Capítulo serão usados no cálculo da exponenciação. O principal objetivo deste Capítulo foi revisar as principais técnicas existentes para a aritmética de precisão múltipla.

Capítulo 3

Criptografia Assimétrica

Neste capítulo é feita uma revisão sobre os principais algoritmos de criptografia assimétrica (chave pública) que utilizam a exponenciação modular assim como os problemas que são as bases das seguranças destes algoritmos criptográficos. Este Capítulo é dividido da seguinte maneira: Seção 3.1 será identificado o Problema do Logaritmo Discreto, Seção 3.2 é introduzido o Problema da Fatoração de Inteiros; Seção 3.3 descreve o algoritmo devido a W. Diffie e M. Helman; Seção 3.4 apresenta o algoritmo RSA e, finalmente, Seção 3.5 introduz o algoritmo denominado ElGamal.

3.1 Problema do Logaritmo Discreto

O Problema do Logaritmo Discreto (PLD) pode ser caracterizado como segue abaixo:

Dado: $m \in \mathbb{N}$ e $g, R \in \mathbb{Z}_m$.

Encontrar: $e \in \mathbb{Z}_m$ tal que $R = g^e \mod m$.

Este problema possui complexidade sub-exponencial (veja Semaev (1998), Adleman e DeMarrais (1994)) com $O(e^{(c+O(1))(\ln m)^{1/3}(\ln \ln m)^{2/3}})$, sendo c um valor constante. Atualmente, para uma segurança aceitável, temos m com, aproximadamente, 1024 bits. Ou seja, esse problema se torna intratável computacionalmente quando $\log_2 m \approx 1024$. Muitos algoritmos criptográficos (veremos alguns nas pró-

Algoritmo	**Complexidade relação a** n
Método das divisões triviais	$\lfloor\sqrt{n}\rfloor$
Método *rho* de Pollard	$\lfloor\sqrt[4]{n}\rfloor$
Médodo de Lenstra	$\exp\left(\sqrt{(1+o(1))\ln n \ln(\ln n)}\right)$
Método das frações contínuas	$\exp\left(\sqrt{2\ln n \ln(\ln n)}\right)$
Algoritmo QS	$\exp\left(\sqrt{\ln n \ln(\ln n)}\right)$
Algoritmo GNFS	$\exp\left(c\sqrt[3]{\ln n}\sqrt[3]{(\ln(\ln n))^2}\right)$

Tabela 3.1: Complexidades dos algoritmos de fatoração de inteiros

ximas seções) utilizam o PLD como base de sua segurança Crandall et al. (1997).

3.2 Problema da Fatoração de Inteiros

O Problema da Fatoração de Inteiros (PFI) pode ser caracterizado como segue abaixo:

Dado: $n \in \mathbb{N}$.

Encontrar: os números primos $p, q \in \mathbb{N}$ tal que $n = p \cdot q$.

Os algoritmos que resolvem o PFI podem ser modificados para resolver o PLD (e *vice-versa*). Deste forma, o PLD e o PFI tem a mesma complexidade computacional. Os principais algoritmos para a fatoração de números inteiros são Number Field Sieve (NFS) Pollard (1988), General Number Field Sieve Buhler et al. (1992), Quadratic Sieve (QS) Pomerance (1985) e Multiple Polynomial Quadratic Sieve (MPQS) Pomerance et al. (1988). Para uma sólida referência no assunto veja Montgomery (1994).

A Tabela 3.1 compara os tempos de execução esperados de alguns dos principais algoritmos de fatoração Montgomery (1994).

3.3 Algoritmo de Diffie-Helman

Whitfield Diffie e Martin E. Hellman Diffie e Hellman (1976) propuseram uma interessante solução para o problema de dois usuários estabelecerem uma chave secreta em um canal de comunicação inseguro, que é considerada a primeira

prática de criptografia de chave pública. A segurança desta solução é baseada na dificuldade computacional de resolver o Problema do Logaritmo Discreto (PLD). O algoritmo de Diffie-Helman foi originalmente descrito como segue: Vamos supor que Alice e Bob queiram estabelecer uma chave criptográfica compartilhada. Neste caso ambos tem conhecimento de um primo p público e do inteiro público g tal que $1 < g < p$.

(1) Alice escolhe um inteiro s_a como a sua chave privada e calcula $K_a = g^{s_a} \mod p$, envia para Bob K_a.

(2) Analogamente, Bob escolhe seu inteiro s_b e envia para Alice $K_b = g^{s_b} \mod p$.

(3) Alice calcula $K_b^{s_a} \mod p$.

(4) Bob calcula $K_a^{s_b} \mod p$.

Observe que $K_b^{s_a} = (g^{s_b})^{s_a} = g^{s_a s_b} = (g^{s_a})^{s_b} = K_a^{s_b} \mod p$. Após a execução deste protocolos, Alice e Bob poderiam utilizar um algoritmo simétrico para trocar informações. É fácil observar que a segurança deste algoritmo está baseada no PLD. Este protocolo pode apresentar algumas falhas de segurança. Na próxima Seção descreveremos uma destas falhas.

3.3.1 Ataque Ativo Man-in-the-Middle

Neste ataque consideraremos um intruso que não apenas lê as informações que trafegam sob um canal, mas como também as altera e injeta. Este caso caracteriza um ataque ativo do tipo *man-in-the-middle* (homem no meio). Em relação ao protocolo descrito na Seção anterior, o intruso poderá se passar tanto por Alice perante Bob quanto por Bob perante Alice. A Figura 3.1 ilustra este aspecto.

Para ludibriar Alice ou Bob o intruso procede da seguinte maneira:

(1) No momento em que Alice envia K_a para Bob, o intruso intercepta e envia outro inteiro, digamos $K_i = g^{s_i} \mod p$. Lembre-se que os inteiros p (primo) e g são públicos.

Figura 3.1: Ataque *man-in-the-middle* ativo.

(2) Bob recebe K_i pensando que, na verdade, é K_a e faz os cálculos de acordo com o algoritmo. Assim Bob estabelece uma chave com o intruso.

(3) Quando Bob envia K_b para Alice, o intruso procede da mesma maneira que no passo 2. E firma uma chave com Alice.

Assim o intruso tem uma chave para se comunicar com Bob e outra para se comunicar com Alice. Para resolver este problema pode-se modificar o algoritmo de maneira que cada usuário utilize como chave pública a tripla (p, g, R) onde $R = g^s \mod p$. Vamos supor que Alice queira enviar w para Bob. Aqui Bob publica previamente sua chave pública (p_b, g_b, R_b). Logo ficamos com a seguinte modificação:

(1) Alice escolhe seu inteiro secreto s_a e calcula $g_b^{s_a} \mod p_b = X_a$ e $K_{ab} = R_b^{s_a} \mod p$. Agora Alice usa K_{ab} para criptografar w: $K_{ab}(w) = u$. Envia u e X_a para Bob.

(2) Bob calcula $X_a^{s_b} \mod p = K_{ab}$ e utiliza esse valor para decriptar u e obtendo assim w

Observe que não existe mais o "dialogo", neste caso só há um envio de dados.

3.4 RSA

O RSA (Ron **R**ivest, Adi **S**hamir e Len **A**dleman) foi publicado em 1978 Rivest et al. (1978) e é atualmente o principal criptossistema de chave pública usado

em aplicações comerciais Coutinho (1997). O Algoritmo RSA tem sua segurança baseada no Problema da Fatoração de Inteiros. Em outras palavras, não se conhece um algoritmo eficaz, para inteiros de tamanho adequado, para fatoração. Hoje, o RSA tem sua segurança aceitável com 1024 bits (alguns bancos utilizam 2048). Inicialmente veremos como funciona o processo de geração de chaves pública/privada (Algoritmo 11).

Algoritmo 11: Geração de chaves RSA.

Saída: A chave pública (n, e) e a chave privada (n, d)

1 **início**
2   Escolher dois primos grandes, digamos p e q;
3   Calcular $n \leftarrow p \cdot q$;
4   Calcular a função $\varphi(n) \leftarrow (p-1)(q-1)$;
5   Escolher um inteiro e tal que $1 < e < \varphi(n)$ e $mdc(e, \varphi(n)) = 1$;
6   Calcular d, tal que $d \cdot e \equiv 1 \mod \varphi(n)$;
7   **retorna** chave pública: (n, e), chave privada: (n, d);
8 **fim**

Para geração de primos grandes olhe o apêndice A. Para criptografar (Algoritmo 12) uma mensagem procedemos da seguinte forma:

Algoritmo 12: Criptografia RSA.

Entrada: A chave pública (n, e) e a mensagem m com $m < n$.
Saída: O texto criptografado c.

1 **início**
2     $c \leftarrow m^e \mod n$;
3   **retorna** o criptograma c;
4 **fim**

O Algoritmo 13 exibe o funcionamento do processo de decifragem de c.

Agora iremos demonstrar o processo de decifragem, ou seja iremos demonstrar o fato que $(m^e)^d = m \mod n$.

Demonstração: Como e é o inverso de d módulo $\varphi(n)$ temos que $d \cdot e = 1 + \varphi(n)k$, para algum $k \in \mathbb{N}$. Assim temos que $m^{de} = m^{1+\varphi(n)k} = (m^{\varphi(m)})^k m \mod n$. Como $\varphi(n) = (p-1)(q-1)$ pelo *Pequeno Teorema de Fermat* temos que:

$$(m^{(p-1)})^{(q-1)} m \equiv m \mod p$$

Algoritmo 13: Decifragem RSA.

Entrada: A chave privada (n, d) e o texto cifrado c.
Saída: A mensagem m.

```
1 início
2 |  m ← c^d mod n;
3 |  retorna a mensagem m;
4 fim
```

se $p \not| m$, (se $p|m$ então $m^{de} \equiv 0 \mod p$). Analogamente podemos demonstrar que $m^{de} \equiv m \mod q$. Como p e q são primos e $m^{de} \equiv m \mod p$ e $m^{de} \equiv m \mod q$ então $m^{de} \equiv m \mod n$. Como $m < n$ a afirmação foi provada. Para consolidar o método desta Seção iremos fazer um exemplo numérico.

Exemplo 3.4.0.1 Iremos criptografar a sigla LNCC, para isso, inicialmente, recorremos à Tabela ASCII para codificar as letras em um único inteiro. Depois de codificada e concatenada, temos que a sigla fica LNCC = 76786767. O próximo passo é a geração de chaves:

Geração de Chaves:

(1) Escolher os primos, neste caso $p = 4294967311$ e $q = 8616319249$.

(2) Calcular $n = 37006809515595069439$.

(3) Calcular $\varphi(37006809515595069439) = 37006809502683782880$.

(4) Escolhemos aleatoriamente $e = 22987520831085250907$, neste caso devemos ter $mdc(e, n) = 1$.

(5) Pelo algoritmo de Euclides Estendido Menezes et al. (1997) temos que $d = 16954631775963686003$.

Nossa chave pública é o par

$$(n, e) = (37006809515595069439, 22987520831085250907)$$

e nossa chave privada fica

$$(n, d) = (37006809515595069439, 16954631775963686003).$$

Para criptografar a mensagem LNCC é necessário o conhecimento da chave pública do destinatário, que neste caso é o par (n, e). O próximo passo é encriptar a mensagem LNCC = 76786767, então

$$76786767^{22987520831085250907} = 34891697997283948788 \mod 37006809515595069439.$$

Neste exemplo o criptograma é o inteiro $c = 34891697997283948788$. Para decriptar c usamos nossa chave privada (n, d) de forma que:

$$34891697997283948788^{16954631775963686003} = 76786767 \mod 37006809515595069439.$$

Recuperando assim a mensagem LNCC = 76786767.

Atualmente recomenda-se, para uma segurança aceitável, chaves de 1024 bits. A empresa americana RSA Labs. oferecia um prêmio em dinheiro quando alguém fatora um número n (produto de dois primos grandes) disponibilizados pela própria empresa. A Tabela 3.2 mostra os números RSA que já foram fatorados. Os dígitos estão em base 10.

3.5 ElGamal

Taher ElGamal ElGamal (1985) propôs um outro algoritmo importante também baseado na dificuldade de resolver o PLD. Este algoritmo também poderá ser utilizado para a assinatura digital. Inicialmente considere o algoritmo 14 para a geração de chaves.

O algoritmo 15 exibe o processo de cifragem do algoritmo ElGamal.

O algoritmo 16 exibe o processo de decifragem o criptossistema ElGamal.

Número RSA	Dígitos	Fatorado em	Algoritmo usado
RSA-100	100	01/04/1991	MPQS
RSA-110	110	14/04/1992	MPQS
RSA-120	120	09/06/1993	MPQS
RSA-129	129	26/04/1994	MPQS
RSA-130	130	10/04/1996	NFS
RSA-140	140	02/02/1999	NFS
RSA-150	150	16/04/2004	GNFS
RSA-155	155	22/08/1999	NFS
RSA-160	160	01/04/2003	GNFS
RSA-576*	174	03/12/2003	GNFS
RSA-640*	193	02/11/2005	GNFS
RSA-200	200	09/05/2005	GNFS

Tabela 3.2: Os números RSA que foram fatorados.

Algoritmo 14: Geração de chaves ElGamal.

Saída: A chave pública e a chave privada

1 **início**
2 Escolher aleatoriamente um primo p e um gerador q do grupo multiplicativo $\mathbb{Z}_p^*$.;
3 Escolher aleatoriamente um número inteiro s tal que $1 \leq s \leq p - 2$.;
4 $T \leftarrow g^s \mod p$;
5 **retorna** a chave pública: (p, g, T), a chave privada s;
6 **fim**

Algoritmo 15: Cifragem ElGamal.

Entrada: A chave pública (p, g, T) e a mensagem $m < p$.
Saída: O texto criptografado (y, z).

1 **início**
2 Escolher aleatoriamente um inteiro k tal que $1 \leq k \leq p - 2$;
3 $y \leftarrow g^k \mod p$;
4 $z \leftarrow m \cdot (T^k) \mod p$;
5 **retorna** o criptograma (y, z);
6 **fim**

Algoritmo 16: Decifragem ElGamal.

Entrada: O criptograma (y, z) e a chave privada s.
Saída: a mensagem m.

1 **início**
2 $X \leftarrow (y^{-1})^s \mod p$;
3 $m \leftarrow X \cdot z \mod p$;
4 **retorna** o texto legível m;
5 **fim**

3.6 Conclusões

Este capítulo revisou três importantes algoritmos de criptografia assimétrica. Nesses algoritmos, a operação aritmética mais importante é a exponenciação modular. Desta forma, uma implementação eficiente da exponenciação modular permite um ganho considerável no desempenho final dos algoritmos explanados nesta Seção.

Capítulo 4

Paralelização para a Exponenciação Modular

Neste capítulo serão apresentadas paralelizações para o algoritmo de exponenciação modular. Inicialmente será proposto um algoritmo paralelo que se baseia na partição da representação binária do expoente. Num segundo momento, serão discutidas mudanças neste algoritmo com o objetivo de usar melhor os recursos paralelos. São analisados os resultados teóricos e práticos que acerca das paralelizações apresentadas.

4.1 Algoritmo Binário para Exponenciação Modular

Este método tem aproximadamente 2000 anos Knuth (1997) e também é conhecido como método da elevação ao quadrado e da multiplicação. A ideia central é calcular $g^e \mod p$ baseada na expansão binária de e. O algoritmo 17 processa os *bits* da esquerda para a direita. Analogamente, o algoritmo 18 processa os *bits* da direita para a esquerda.

Exemplo: Tomando $e = 22 = (10110)_2$

Algoritmo 17					
e	1	0	1	1	0
a	g^1	g^2	g^5	g^{11}	g^{22}

Algoritmo 18					
e	1	0	1	1	0
a	g^{22}	g^6	g^6	g^2	g^0

Também referenciado com *square-and-multiply* o algoritmo binário de expo-

Algoritmo 17: Algoritmo binário versão esquerda para direita.

Entrada: Inteiro $g \in \mathbb{Z}_p$ e $e = (b_n \dots b_2 b_1)_2$ onde $b_i \in \{0, 1\}$ (representação em base binária).

Saída: $g^e \mod p$.

```
1 início
2     a ← 1;
3     para i = n até 1 faça
4         a ← a^2 mod p;
5         se b_i = 1 então
6             a ← a · g mod p;
7     retorna a;
8 fim
```

Algoritmo 18: Algoritmo binário versão direita para esquerda.

Entrada: Inteiro $g \in \mathbb{Z}_p$ e $e = (b_n \dots b_2 b_1)_2$ onde $b_i \in \{0, 1\}$ (representação em base binária).

Saída: $g^e \mod p$.

```
1 início
2     a ← 1;
3     para i = 1 até n faça
4         se b_i = 1 então
5             a ← a · g mod p;
6         g ← g^2 mod p;
7     retorna a;
8 fim
```

nenciação modular pode ser descrito recursivamente como:

$$g^e = \begin{cases} 1 & \text{se } e = 0 \\ (g^{e/2})^2 & \text{se } e \text{ é par} \\ g^{e-1}g & \text{se } e \text{ é ímpar} \end{cases}$$

A complexidade computacional de ambos algoritmos binários de exponenciação modular é n elevações ao quadrado e $\frac{n}{2}$ multiplicações modulares em média, onde n é o número de *bits* do expoente e.

4.2 Paralelização do Algoritmo Binário para um Número Pequeno de Processadores

Quando levamos em conta a representação em base binária do expoente $e = (b_n b_{n-1} \ldots b_1 b_1)_2$, podemos identificar a seguinte identidade

$$g^e \mod n = g^{2^{r+1}e_2} \cdot g^{e_1} \mod n, \tag{4.1}$$

onde $e_1 = (b_r \ldots b_1 b_1)_2$, $e_2 = (b_n \ldots b_{r+1})_2$ e $r = \lceil r/2 \rceil$. Na verdade, a equação (4.1) segue do fato que:

$$e = 2^r e_2 + e_1. \tag{4.2}$$

Isto posto, podemos obter os subprodutos modulares $g^{2^r e_2}$ e g^{e_1} de (4.1) em paralelo. A equação (4.1) resume a idéia central da paralelização do algoritmo binário de exponenciação. O algoritmo 19 sistematiza esta idéia.

Quando n (número de *bits* de e) for par, a "partição" do expoente e será igualmente dividida, ou seja, e_1 e e_2 ficam, cada um, com $\frac{n}{2}$ *bits* em sua representação. Senão, e_1 deverá ter um *bit* a mais em relação a e_2. Como a computação na Região Paralela 2 (veja algoritmo 19) é mais custosa que na Região Paralela 1, é mais sensato, quando n for impar, que e_1 tenha um *bit* a mais que e_2. Esta escolha pode permitir um equilíbrio maior entre o tempo de execução entre as regiões paralelas, o que é desejado. O algoritmo 19 atua com duas linhas de execução paralelas, no

Algoritmo 19: Paralelização do algoritmo binário.

Entrada: Inteiro $g \in \mathbb{Z}_p$ e $e = (b_n \dots b_2 b_1)_2$ onde $b_i \in \{0, 1\}$ (representação em base binária).

Saída: $g^e \mod p$.

1 **início**
2 $e_1 \leftarrow (b_r \dots b_1 b_1)_2$;
3 $e_2 \leftarrow (b_n \dots b_{r+1})_2$;
4 **início**
5 **[Região Paralela 1]**
6 $a_1 \leftarrow g^{e_1} \mod p$ (algoritmo 17 ou 18);
7 **fim**
8 **início**
9 **[Região Paralela 2]**
10 $a_2 \leftarrow g^{2^r}$;
11 $a_2 \leftarrow a_2^{e_2} \mod p$ (algoritmo 17 ou 18);
12 **fim**
13 **retorna** $a_1 \cdot a_2 \mod p$;
14 **fim**

entanto, essa idéia pode ser mais geral - poderíamos ter $e_1, e_2, \ldots, e_k$ onde k é o número de vias paralelas de execução e $r_1 = 0, r_2 = \frac{n}{k}, r_3 = \frac{2n}{k}, \ldots, r_k = \frac{(k-1)n}{k}$. Desta forma, poderíamos proceder de forma análoga ao que foi mostrado no algoritmo 19 (veja o algoritmo 20).

No algoritmo 20, o tamanho do expoente e_i para $i = 1, \ldots, k$ também deve ser levado em conta. Nesse caso, o resto da divisão do número de *bits* de e por k de ser analisado.

4.2.1 Análise de Complexidade

Voltando ao algoritmo 20, é fácil perceber que a Região Paralela 2 do algoritmo 19 exige uma computação maior. Neste caso, quando computamos $a_2 = g^{2^r} \bmod p$ na verdade estamos calculando r elevações ao quadrado e 1 multiplicação (veja algoritmo 17 ou 18). E, finalmente, quando é calculado $a_2 = a_2^{e_2}$ fazemos r elevações ao quadrado e $\frac{r}{2}$ multiplicações em média. Assim, a Região Paralela 2 do algoritmo 19 consome, aproximadamente, $2r$ elevações ao quadrado e $\frac{r}{2}$ multiplicações. Se somarmos a multiplicação computada na linha 13 do algoritmo 19, temos um custo computacional de

$$2rE + \left(\frac{r}{2} + 1\right) M, \tag{4.3}$$

onde M é o tempo requerido para uma multiplicação e E é o tempo computacional para uma elevação ao quadrado. Essa análise vale quando k é par, que é o pior caso. Já em função de n, a expressão (4.3) fica

$$nE + \left(\frac{n}{4} + 1\right) M. \tag{4.4}$$

Observe, que não incluímos a Região Paralela 1 no tempo computacional. Isso segue do fato que esta região possui um custo computacional médio de

$$rE + \frac{r}{2}M,$$

Algoritmo 20: Paralelização com n linhas de execução.

Entrada: Inteiro $g \in \mathbb{Z}_p$ e $e = (b_n \dots b_2 b_1)_2$ onde $b_i \in \{0, 1\}$ (representação em base binária).

Saída: $g^e \mod p$.

```
1  início
2  |  e_1 ← (b_{r_2} ... b_1 b_1)_2;
3  |  e_2 ← (b_{r_3} ... b_{r_2+1})_2;
4  |  ⋮
5  |  e_k ← (b_j ... b_{r_k+1})_2;
6  |  início
7  |  |  [Região Paralela 1]
8  |  |  a_1 ← g^{e_1} mod p;
9  |  fim
10 |  início
11 |  |  [Região Paralela 2]
12 |  |  a_2 ← g^{2^{r_2}} mod p;
13 |  |  a_2 ← a_2^{e_2} mod p;
14 |  fim
15 |  ⋮
16 |  início
17 |  |  [Região Paralela k]
18 |  |  a_k ← g^{2^{r_k}} mod p;
19 |  |  a_k ← a_k^{e_k} mod p;
20 |  fim
21 |  retorna a_1 · a_2 · ... · a_k mod p;
22 fim
```

ou seja, faz rE a menos que a Região Paralela 2. Comparando a complexidade do algoritmo binário sequencial, que é de $nE + \left(\frac{n}{2}\right) M$, com a complexidade do algoritmo paralelo (veja expressão (4.4)), temos uma diferença de $\frac{n}{4}M$ aproximadamente.

Estas idéias poderão ser generalizadas para k linhas paralelas. Para isso basta tomar $r = \frac{n}{k}$ e lembrar que, ao final do algoritmo (linha 21 do algoritmo 20), $k-1$ multiplicações são executadas. O número de elevações ao quadrado continua sendo n, no entanto, o número de multiplicações fica $\frac{n}{2k} + k - 1$. Generalizando, temos um custo computacional de:

$$nE + \left(\frac{n}{2k} + k - 1\right) M. \tag{4.5}$$

Assim, em todos os casos sempre gastamos nE. A diferença fica no número de multiplicações. Desta forma, para fazer uma análise um pouco mais detalhada precisamos explorar a função $\eta(n, k)$ que retorna no número de multiplicações definida por

$$\eta(n, k) = \frac{n}{2k} + k - 1. \tag{4.6}$$

A fim de minimizar o número de multiplicações e fazendo n constante, considere a derivada parcial

$$\frac{\partial \eta(n, k)}{\partial k} = -\frac{n}{2k^2} + 1. \tag{4.7}$$

Igualando (4.7) a zero e resolvendo a equação na variável k, temos a seguinte relação entre o número de linhas paralelas e o número de *bits* do expoente

$$k = \sqrt{\frac{n}{2}}. \tag{4.8}$$

Como a concavidade de $\eta(n, k)$ é positiva, segue que a equação (4.8) fornece um mínimo global. Este é o número ótimo de linhas paralelas k em função do número de *bits* n, que determina a segurança do criptossistema.

4.3 Paralelização Massiva

O algoritmo 20 não usa totalmente os recursos paralelos. A grande idéia do próximo algoritmo é paralelizar a computação da linha 21 do algoritmo 20 de forma que cada processador multiplique em paralelo o resultado parcial. Assim, na primeira iteração, usamos $\frac{k}{2}$ processadores sendo que cada processador $i = \{1, \ldots, \frac{k}{2}\}$ computa $a_i = a_{2i-1} \cdot a_{2_i}$. Na próxima iteração será necessário $\frac{k}{4}$ processadores, desta forma precisamos de $\lceil \log_2 k \rceil$ iterações. O algoritmo seguinte descreve a execução paralela para a linha 21 do algoritmo 20.

Algoritmo 21: Paralelização da linha 21 do algoritmo 20.

Entrada: Inteiros $a_1, a_2, \ldots, a_k \in \mathbb{Z}_p$

Saída: O produtório $\prod_{i=1}^{k} a_i \in \mod p$.

```
1 início
2   enquanto k ≠ 1 faça
3     para cada processador i = 1 até k/2 faça em paralelo
4       A_i ← a_{2i−1} · a_{2i} mod p;
5     k ← k/2;
6     para i = 1 até k faça
7       a_i ← A_i;
8   retorna A_1;
9 fim
```

4.3.1 Análise de Complexidade

A diferença dos dois algoritmos apresentados está no número de multiplicações modulares. Neste caso, a linha 21 do primeiro algoritmo executa $n - 1$ multiplicações modulares. Com a modificação apresentada nesta seção a linha 21 passa a computar $\lceil \log_2 k \rceil$ multiplicações, assim o custo computacional para este algoritmo fica

$$nE + \left(\frac{n}{2k} + \lceil \log_2 k \rceil\right) M. \tag{4.9}$$

Neste caso, o número de multiplicações modulares é

$$\mu(n, k) = \frac{n}{2k} + \lceil \log_2 k \rceil. \tag{4.10}$$

Desta forma, resolvendo $\frac{\partial \mu(n,k)}{\partial k} = 0$ temos:

$$k = \frac{\ln 2}{2} n \tag{4.11}$$

Na expressão 4.5, que nos dava o custo computacional para o algoritmo 20, temos um total de multiplicações iguais a

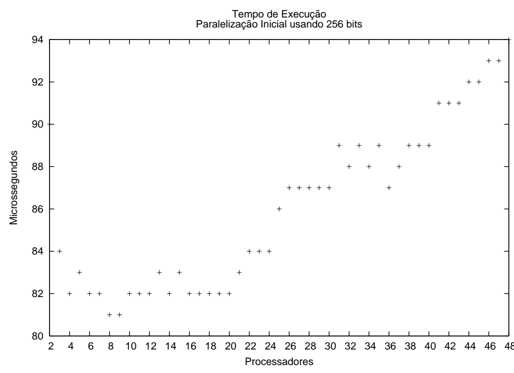
$$\eta(n,k) = \frac{n}{2k} + k - 1. \tag{4.12}$$

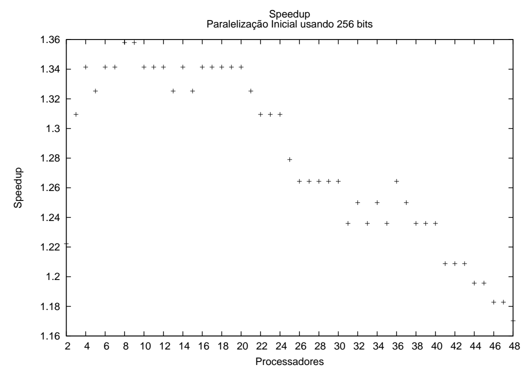
Se compararmos com a equação 4.10, reduzimos um fator linear $(k-1)$ a um fator logarítmico $(\lceil \log_2 k \rceil)$ no número de multiplicações modular.

4.4 Resultados

A implementação foi desenvolvida em linguagem `C` usando como ferramenta básica de paralelização a biblioteca MPICH2 que implementa o padrão MPI (Message Passing Interface) Snir et al. (1998). A API (Application Programming Interface) do MPI oferece, de maneira eficaz, a paralelização usando memória física distribuída. Para aritmética de precisão múltipla utilizamos a biblioteca GMP (GNU Multiple Precision) Granlund (2002). As escolhas acima visaram um bom desempenho de tempo de execução. Os recursos de *hardware* utilizados foram 6 nós `Sun Blade x6250` cada nó com 2 processadores `Intel Xeon E5440 Quad Core 3GHz` e 16GB de memória física interligados por um barramento *InfiniBand*. Para todos os testes de desempenho, utilizamos um expoente cuja esparsidade da representação binária seja $\frac{1}{2}$. Neste caso, o expoente para os testes ficaram da forma $(101010\ldots1010)_2$. Para os testes, usamos até 48 processadores. As figuras de 4.1 a 4.3 exibem as medianas do tempo de execução e os *speedups* de 100 coletas sucessivas para até 48 processadores para 256, 512 e 1024 bits no expoente referentes ao algoritmo 20. A escala do tempo de execução está dada em microssegundos.

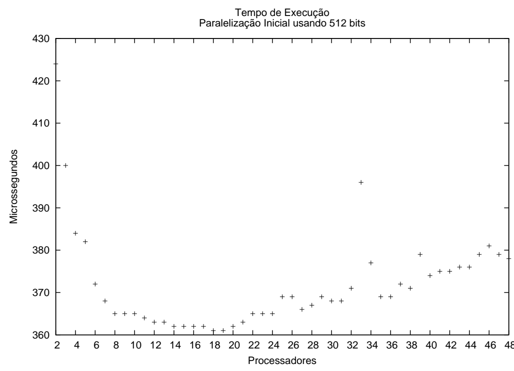
Para expoentes de 512 bits, por exemplo, o *speedup* começa a se degenerar a partir de 20 processadores. Considerando que da análise teórica temos um número

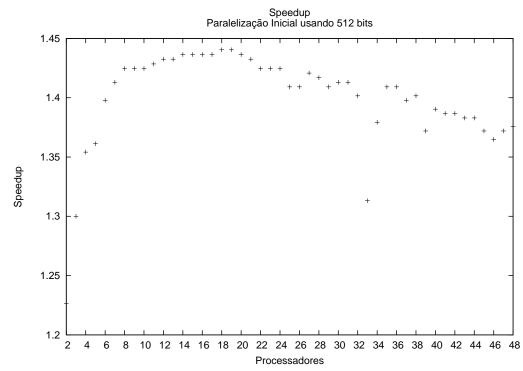




(a) (b)

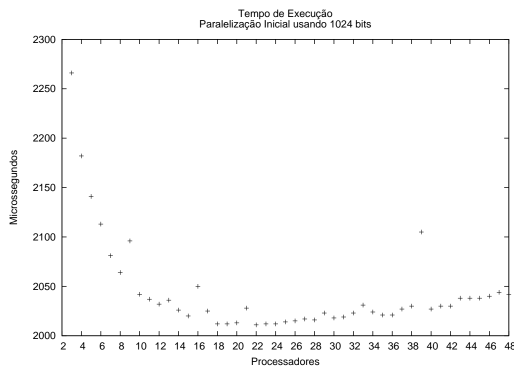
Figura 4.1: Tempo de execução (a) e *speedup* (b) para até 48 processadores usando 256 bits

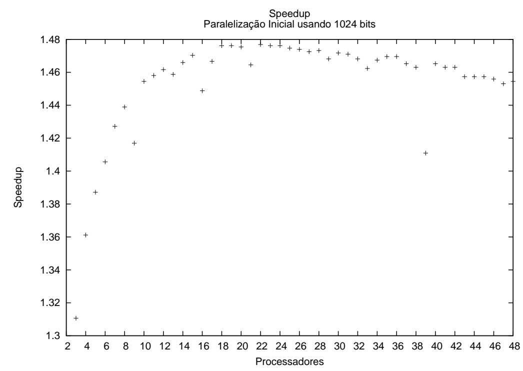




(a) (b)

Figura 4.2: Tempo de execução (a) e *speedup* (b) para até 48 processadores usando 512 bits





(a) (b)

Figura 4.3: Tempo de execução (a) e *speedup* (b) para até 48 processadores usando 1024 bits

ótimo de processadores igual $n = \sqrt{\frac{j+1}{2}}$, onde $(j+1)$ representa o número de bits, neste caso, para 512 bits, temos $n = \sqrt{\frac{512}{2}} = 16$, que é relativamente próximo do

obtido (ver figura 4.2). A figura de 4.4 exibe o desempenho para o algoritmo 21 (paralelização massiva) usando 1024 bits.

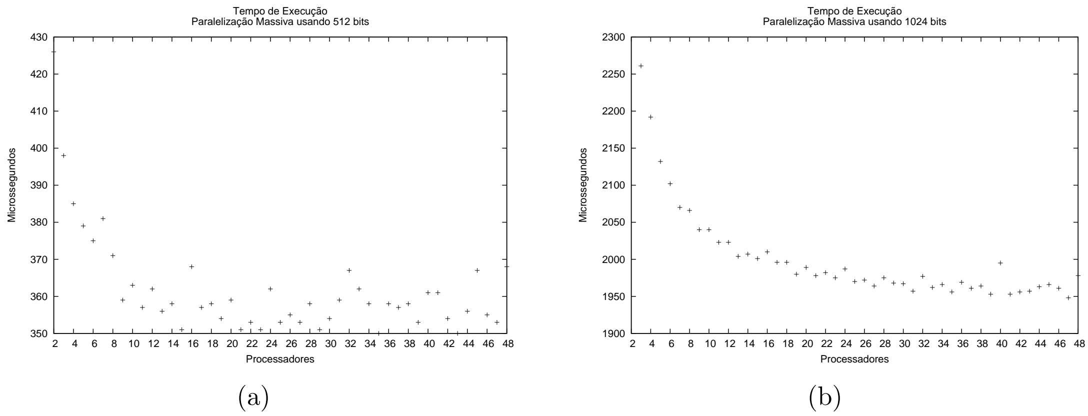


Figura 4.4: Desempenho experimental para o algoritmo 21. (a) usando 512 bits (b) 1024 bits

Para o algoritmo 21, não foi possível obter um mínimo global devido as restrições de número de processador utilizado no experimento. No entanto, os testes comprovaram os estudos teóricos com relação ao tempo computacional.

A figura 4.5 compara os algoritmos 20 e 21 para entradas de 1024 bits. Para até 20 processadores quase não existe diferença entre os algoritmos. O *overhead* gerado pela comunicação entre os processos no algoritmo 21 faz com que ele não seja mais rápido que o algoritmo 20 para até 20 processadores. No gráfico da figura 4.5 é possível observar que a implementação do algoritmo 21 se mantém escalável e com o tempo de execução inferior ao gráfico do algoritmo 20 a partir de 20 processadores. O gráfico da figura 4.5 permite distinguir claramente aspectos importantes dos dois métodos: o algoritmo 20 atua bem com poucos processadores e necessita de pouca comunicação entre os processos. A modificação usando o algoritmo 21, por sua vez, exige muita comunicação entre os processos e é substancialmente mais escalável que o algoritmo 20. No gráfico da figura 4.5, o algoritmo 21 não apresenta uma concavidade, neste caso o mínimo e máximo estão localizados nos extremos. Da análise teórica, que resulta na equação (4.8), temos um mínimo de aproximadamente 23 processadores, ficando coerente com os resultados experimentais do algoritmo 20 apresentado na figura 4.5.

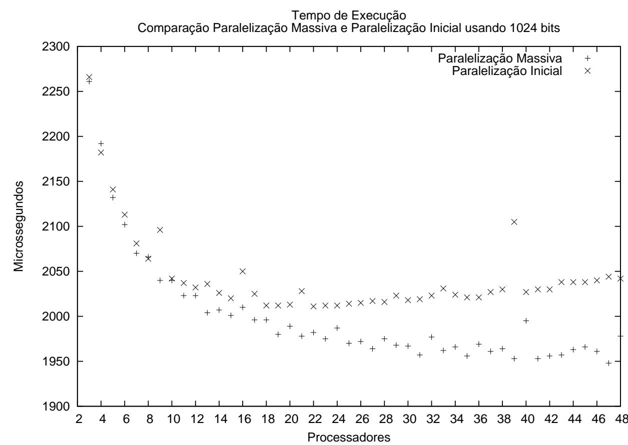


Figura 4.5: Comparativo entre os algoritmo de paralelização apresentados.

4.5 Conclusões

Este capítulo descreveu propostas para a paralelização o algoritmo da exponenciação modular, usado na cifragem e decifragem do método de criptografia RSA. Inicialmente foi apresentado um algoritmo paralelo baseado na partição do expoente. Posteriormente foi observado que o algoritmo apresentado não usava todos os recursos paralelos disponíveis. Desta forma, foi apresentadas modificações no primeiro algoritmo para conceber um algoritmo mais eficiente. Foi obtido uma redução significativa do número de multiplicações modulares no desempenho final do algoritmo de exponenciação. Anteriormente tínhamos um fator linear $(k-1)$ e com a modificação temos um fator logarítmico $(\log_2(k))$ no número de multiplicações necessárias, onde k é o número de processadores. Assim a nova paralelização se mantém escalável para um número substancialmente maior de processadores. Este fato motiva, como trabalhos futuros, explorar o poder das unidades gráficas de processamento, GPU, para acelerar o processo de exponenciação modular. Em aplicações reais, isto pode ser bastante interessante para servidores com altas cargas de requisições de segurança, por exemplo, *https servers*. Também, é de grande interesse, estudar a implementação de algoritmos para a multiplicação por escalar em curvas elípticas, visto que técnicas criptográficas baseadas em curvas elípticas apresentam, de maneira geral, uma chave substancialmente menor que os algorit-

mos baseados no problema do logaritmo discreto em $\mathbb{Z}_p$. É de grande relevância ressaltar que a classe de algoritmos para a multiplicação por escalar em curvas elípticas possui relação bijetiva com os algoritmos de exponenciação modular.

Capítulo 5

Balanceamento de Carga

5.1 Balanceamento

A carga entre os processadores do algoritmo 19, no caso médio, não está devidamente balanceada. Para observar isto, basta verificar o custo computacional entre os processadores. Para a Região Paralela 1 (algoritmo 19) temos o seguinte custo computacional:

$$T_1 = \left(\frac{n}{2}\right) E + \left(\frac{n}{4}\right) M,$$

e o custo computacional para a Região Paralela 2 é dado por

$$T_2 = nE + \left(\frac{n}{4}\right) M.$$

A diferença entre as regiões é dado por

$$T_2 - T_1 = \left(\frac{n}{2}\right) E$$

Ou seja, a Região Paralela 2 computa, no caso médio, $\frac{n}{2}$ elevações ao quadrado a mais que a Região Paralela 1. Agora, ao invés de ponto de partição central $\frac{n}{2}$, considere um ponto de partição p, tal que $T_1 = T_2$. Assim os custos computacionais entre os processadores ficam

$$T_1 = pE + \frac{p}{2}M$$

$$T_2 = nE + \left(\frac{n-p}{2}\right) M$$

Para facilitar os cálculos consideramos E igual a uma unidade de tempo e introduzimos uma constante φ, tal que $\varphi = \frac{M}{E}$. Igualando os tempos computacionais $T_1 = T_2$ temos a seguinte equação

$$p + \frac{p}{2}\varphi = n + \left(\frac{n-p}{2}\right)\varphi.$$

Resolvendo na variável p temos o seguinte resultado

$$p = \frac{n(1+\frac{\varphi}{2})}{1+\varphi}.$$

Para o caso geral (para k processadores) temos a seguinte consideração: precisamos encontrar a sequência de pontos de partição $\{p_1, p_2, \ldots, p_k\}$ tal que $p_k = n$ e este seja os pontos de partições que igualam o processamento entre os processadores $(T_1 = T_2 = T_3 = \ldots = T_k)$. Desta forma temos as seguintes equações:

$$\begin{aligned}
&T_1 = p_1 + \varphi(\tfrac{p_1}{2}) = p_1(1+\tfrac{\varphi}{2})\\
&T_2 = p_2 + \varphi(\tfrac{p_2-p_1}{2}) = p_2(1+\tfrac{\varphi}{2}) - \tfrac{\varphi}{2}p_1\\
&T_3 = p_3(1+\tfrac{\varphi}{2}) - \tfrac{\varphi}{2}p_2\\
&T_4 = p_4(1+\tfrac{\varphi}{2}) - \tfrac{\varphi}{2}p_3\\
&\vdots\\
&T_i = p_i(1+\tfrac{\varphi}{2}) - \tfrac{\varphi}{2}p_{i-1}\\
&\vdots\\
&T_k = p_k(1+\tfrac{\varphi}{2}) - \tfrac{\varphi}{2}p_{k-1}
\end{aligned}$$

Explorando o termo geral $T_i = T_{i-1}$ para $i > 1$ temos

$$p_i(1+\frac{\varphi}{2}) - \frac{\varphi}{2}p_{i-1} = p_{i-1}(1+\frac{\varphi}{2}) - \frac{\varphi}{2}p_{i-2}$$

$$p_i(1+\frac{\varphi}{2}) = p_{i-1}(1+\varphi) - \frac{\varphi}{2}p_{i-2}$$

De maneira geral temos a seguinte relação de recorrência linear homogênea de grau dois

$$p_i = p_{i-1}\frac{1+\varphi}{1+\frac{\varphi}{2}} - p_{i-2}\frac{\varphi}{2(1+\frac{\varphi}{2})}$$

Considerando as condições iniciais

$$p_k = n \tag{5.1}$$

$$p_1 = p_2\left(\frac{1+\frac{\varphi}{2}}{1+\varphi}\right). \tag{5.2}$$

Uma solução geral para a recorrência acima pode ser dada por

$$p_i = \alpha x_1^i + \beta x_2^i$$

onde x_1, x_2 são raízes do polinômio característico

$$\rho(x) = x^2 - x\frac{1+\varphi}{1+\frac{\varphi}{2}} + \frac{\varphi}{2(1+\frac{\varphi}{2})}$$

cuja as raízes são $x_1 = 1, x_2 = \frac{\varphi}{2+\varphi}$. Considerando as condições iniciais (5.1) e (5.2) temos o seguinte sistema de equações:

$$\begin{cases} \alpha + \beta\frac{\varphi}{2+\varphi} = \frac{1+\frac{\varphi}{2}}{1+\varphi}\left(\alpha + \beta\left(\frac{\varphi}{2+\varphi}\right)^2\right) \\ \alpha + \beta\left(\frac{\varphi}{2+\varphi}\right)^k = n \end{cases} \tag{5.3}$$

Da primeira equação de 5.3 temos que $\alpha = -\beta$. Substituindo β na segunda equação temos:

$$\alpha = \frac{n}{1-\left(\frac{\varphi}{2+\varphi}\right)^k}, \tag{5.4}$$

$$\beta = \frac{n}{\left(\frac{\varphi}{2+\varphi}\right)^k - 1}. \tag{5.5}$$

Assim

$$p_i = \alpha \left(1 - \lambda^i\right), \tag{5.6}$$

onde,

$$\lambda = \frac{\varphi}{2 + \varphi}.$$

Mas essa solução geral só é válida se tivermos uma sequência de pontos de partição crescentes, ou seja, $p_1 < p_2 < \ldots < p_k$. Para verificar o crescimento da sequência p_i em relação ao número de processadores k poderemos verificar o comportamento da sua derivada parcial em relação a i. (considerando o número de processadores k e o número de *bits* do expoente n constantes)

$$\frac{\partial p_i}{\partial i} = -\alpha \, \lambda^i \, \log \lambda.$$

Considerando $n > 2$ e $\log \lambda < 0$ segue que $\frac{\partial p_i}{\partial i} > 0$. Assim a sequência descrita por 5.6 é sempre crescente. Em termos práticos os valores da sequência descrita por 5.6 serão arrendondados para o inteiro mais próximo. Assim dois pontos vizinhos, p_i e p_{i+1} podem ser arredondados para o mesmo valor se a diferença entre eles for menor que $1/2$. Podemos garantir esta restrição explorando a diferença entre os termos

$$p_i - p_{i-1} > \frac{1}{2}.$$

Para todo i. Usando o fato que a primeira derivada $\frac{\partial p_i}{\partial i}$ é positiva e a segunda derivada $\frac{\partial^2 p_i}{\partial i^2}$ é negativa para todo i, nós podemos concluir que a sequência de pontos de partição é monótona e o intervalo $p_{i+1} - p_i$ é decrescente quando i cresce. Desta forma, só precisamos verificar $p_i - p_{i-1} > \frac{1}{2}$. Usando as equações 5.1 e 5.6 nós obtemos

$$p_k - p_{k-1} = n - \alpha(1 - \lambda^{k-1}).$$

Usando a equação 5.4, a restrição $p_i - p_{i-1} > \frac{1}{2}$ pode ser escrita por

$$n > \frac{1}{2\lambda^{k-1}} \left(\frac{1 - \lambda^k}{1 - \lambda}\right). \tag{5.7}$$

Usando a equação 5.7 e explorando a variação k e $k+1$ nós obtemos

$$\frac{1-\lambda^k}{2\lambda^{k-1}(1-\lambda)} < n < \frac{1-\lambda^{k+1}}{2\lambda^k(1-\lambda)}. \tag{5.8}$$

Depois de isolar a variável k, nós temos

$$\gamma < k < \gamma + 1, \tag{5.9}$$

onde

$$\gamma = \log_\lambda \frac{1}{1+2n(1-\lambda)}. \tag{5.10}$$

Assim podemos obter o número ótimo de processadores k dado o intervalo 5.9.

$$k = \left[\gamma + \frac{1}{2}\right], \tag{5.11}$$

A Figura 5.1 exibe o número ótimo de processadores em função do número de bits do expoente e. Observe que o número ótimo de processadores k possui dependência logarítmica no número de bits n.

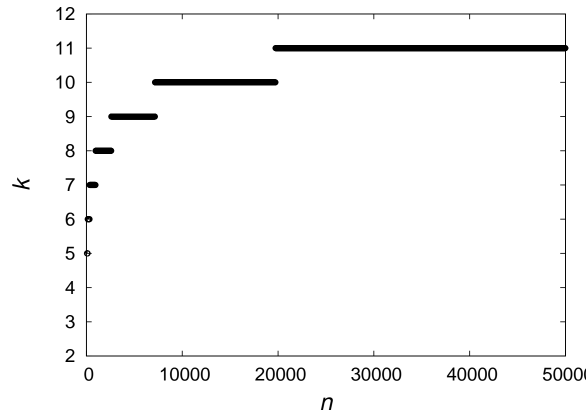


Figura 5.1: Número ótimo de processadores k em função do número de bits n do expoente.

onde a notação [ ] denota o arredondamento para o inteiro mais próximo.

Exemplo: *Considerando a razão* $\varphi = 1.14$ *e entradas de* $n = 1024$ *bits. O número de processadores que melhor satisfaz a inequação 5.7 é* $k = 8$*, uma vez que*

$$\frac{1}{2}\left(1 - \frac{1-\left(\frac{\varphi}{2+\varphi}\right)^7}{1-\left(\frac{\varphi}{2+\varphi}\right)^8}\right)^{-1} = 943.8699$$

$$\frac{1}{2}\left(1 - \frac{1-\left(\frac{\varphi}{2+\varphi}\right)^8}{1-\left(\frac{\varphi}{2+\varphi}\right)^9}\right)^{-1} = 2600.2819$$

Dessa forma, os pontos de partições que melhor balanceiam a carga entre os processadores são

$$p_1 = 652.42623896$$

$$p_2 = 889.294363933$$

$$p_3 = 975.291071726$$

$$p_4 = 1006.5128064$$

$$p_5 = 1017.84808587$$

$$p_6 = 1021.96344211$$

$$p_7 = 1023.45755234$$

$$p_8 = 1024.0$$

Se ao invés de $k = 8$*, tomarmos* $k = 9$ *temos a seguinte sequência de pontos de partição*

$$p_1 = 652.300785978$$

$$p_2 = 889.123364326$$

$$p_3 = 975.103536083$$

$$p_4 = 1006.31926723$$

$$p_5 = 1017.65236707$$

$$p_6 = 1021.76693199$$

$$p_7 = 1023.26075492$$

$$p_8 = 1023.80309827$$

$$p_9 = 1024.0$$

Na sequência final não é crescente, os termos $[p_8]$ e $[p_9]$ assumem o mesmo valor.

5.2 Análise Experimental

As implementações das técnicas foram desenvolvidas em linguagem `C` usando como ferramenta básica de paralelização a biblioteca MPICH2 que implementa o padrão MPI (Message Passing Interface) Snir et al. (1998). A API (Application Programming Interface) do MPI oferece, de maneira eficaz, a paralelização usando memória física distribuída. Para aritmética de precisão múltipla utilizamos a biblioteca GMP (GNU Multiple Precision) Granlund (2002). O *hardware* utilizado foram 6 nós `Sun Blade x6250` cada nó com 2 processadores `Intel Xeon E5440 Quad Core 3GHz x86 64` (totalizando 48 *cores*) e 16GB de memória física interligados por um barramento *InfiniBand*. O sistema operacional utilizado foi o CentOS Linux 4.1.2 e a versão kernel 2.6.18. Para todos os testes de desempenho utilizamos um expoente cuja a esparsidade da representação binária seja $\frac{1}{2}$. Neste caso, o expoente para os testes ficaram da forma $(101010\ldots1010)_2$. A unidade de tempo usada foi o microssegundo.

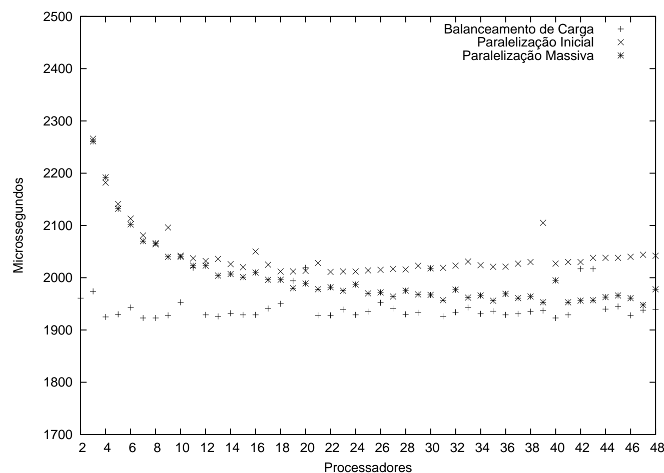


Figura 5.2: Comparação entre os três algoritmos de paralelização apresentados.

No ambiente utilizado, após 1000 amostras usando a biblioteca de precisão múltipla GMP, a razão entre o tempo de uma multiplicação modular e o tempo de uma elevação ao quadrado ficou $\varphi = 1.14$. As comparações entre os métodos podem ser visualizados na figura 5.2. A técnica que usa balanceamento de carga se mostrou mais eficiente que as técnicas sem o balanceamento de carga. No entanto, o grande ganho do método que usa o balanceamento de carga está na redução significativa do número de processadores. Nos experimentos realizados, a técnica com balanceamento de carga teve o tempo mínimo em 8 processadores conforme esperado da análise teórica.

5.3 Conclusão

Este trabalho propôs uma técnica de balanceamento de carga para acelerar a paralelização do algoritmo de exponenciação modular. Inicialmente foi apresentado um algoritmo paralelo para a computação da exponenciação. Foi identificado que o algoritmo apresentado inicialmente não usava totalmente os recursos paralelos disponíveis. Desta forma, foi proposto um modelo de balanceamento de carga baseado neste algoritmo para o cálculo da exponenciação modular. Foram explorados os resultados teóricos que nortearam a implementação experimental da técnica proposta. Por exemplo, para expoentes de 1024 bits e o coeficiente $\varphi = 1.14$ temos um limite superior de 8 processadores. Neste caso, se usarmos mais de 8 processadores teremos pontos de partições redundantes, $[p_k] = [p_{k-1}]$. Os resultados práticos alcançados refletiram os estudos teóricos de maneira satisfatória. Na prática, servidores de alta carga, por exemplo *https servers*, poderão obter o coeficiente φ e determinar dinamicamente o número de ótimo de processadores. A redução significativa do número de processadores pode ser considerado o grande resultado mostrado neste trabalho. O uso de balanceamento de carga se mostrou muito promissor com relação a paralelização sem o balanceamento de carga. Em todos os casos, o método proposto neste trabalho se mostrou superior ao algoritmo sem o uso de balanceamento de carga.

Referências Bibliográficas

Leonard M. Adleman e Jonathan DeMarrais. A subexponential algorithm for discrete logarithms over all finite fields. In: **CRYPTO '93: Proceedings of the 13th Annual International Cryptology Conference on Advances in Cryptology**, páginas 147–158, London, UK, 1994. Springer-Verlag. ISBN 3-540-57766-1.

Paul Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In: **Proceedings on Advances in cryptology—CRYPTO '86**, páginas 311–323, London, UK, 1987. Springer-Verlag. ISBN 0-387-18047-8.

Ernest F. Brickell. A survey of hardware implementation of RSA. In: **CRYPTO '89: Proceedings of the 9th Annual International Cryptology Conference on Advances in Cryptology**, páginas 368–370, London, UK, 1990. Springer-Verlag. ISBN 3-540-97317-6.

Ernest F. Brickell, Daniel M. Gordon, Kevin S. Mccurley, e David B. Wilson. Fast exponentiation with precomputation. In: **Advances in Cryptology – Proceedings of CRYPTO'92**, volume 658, páginas 200–207. Springer-Verlag, 1992.

Ernest F. Brickell, Daniel M. Gordon, Kevin S. Mccurley, e David B. Wilson. Fast exponentiation with precomputation: Algorithms and lower bounds, 1995. Preprint http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.5.606.

J. P. Buhler, H. W. Lenstra, e C. Pomerance. Factoring integers with the number field sieve. 1992.

Severino Collier Coutinho. **Números Inteiros e Criptografia RSA**. IMPA, Rio de Janeiro, RJ, Brasil, 1997.

Richard Crandall, Karl Dilcher, e Carl Pomerance. A search for Wieferich and Wilson primes. **Math. Comp.**, 66(217):433–449, 1997. ISSN 0025-5718. URL http://dx.doi.org/10.1090/S0025-5718-97-00791-6.

Tom St Denis. **BigNum Math: Implementing Cryptographic Multiple Precision Arithmetic**. Syngress Publishing, 2006. ISBN 1597491128.

Whitfield Diffie e Martin E. Hellman. New directions in cryptography. **IEEE Trans. Information Theory**, IT-22(6):644–654, 1976. ISSN 0018-9448.

Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. **IEEE Trans. Inform. Theory**, 31(4):469–472, 1985. ISSN 0018-9448.

Torbjörn Granlund. GNU multiple precision arithmetic library 4.1.2, December 2002. http://swox.com/gmp/.

Anatoly A. Karatsuba e Y. Ofman. Multiplication of multidigit numbers on automata. **Soviet Physics Doklady**, 7:595–596, 1963. ISSN 0038–5689. URL: http://cr.yp.to/bib/entries.html#1963/karatsuba.

Donald E. Knuth. **Art of Computer Programming, Volume 2: Seminumerical Algorithms (3rd Edition)**. Addison-Wesley Professional, November 1997. ISBN 0201896842.

Pedro C. S. Lara, Fábio Borges, e Renato Portugal. Paralelização eficiente para o algoritmo binário de exponenciação modular. In: **Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais**, Campinas - SP, 2009. URL http://sbseg2009.inf.ufsm.br/sbseg2009/.

Chae Hoon Lim e Pil Joong Lee. More flexible exponentiation with precomputation. In: **Advances in Cryptology – CRYPTO'94**, páginas 95–107, 1994.

Alfred J. Menezes, Paul C. Van Oorschot, Scott A. Vanstone, e R. L. Rivest. Handbook of applied cryptography, 1997.

Gary L. Miller. Riemann's hypothesis and tests for primality. In: **Seventh Annual ACM Symposium on Theory of Computing (Albuquerque, N.M., 1975)**, páginas 234–239. Assoc. Comput. Mach., New York, 1975.

P. Montgomery. A survey of modern integer factorization algorithms, 1994. URL `citeseer.ist.psu.edu/montgomery94survey.html`.

Peter L. Montgomery. Modular multiplication without trial division. **Mathematics of Computation**, 44(170):pp. 519–521, 1985. ISSN 00255718.

N. Nedjah e L. M. Mourelle. Efficient parallel modular exponentiation algorithm. In: **ADVIS '02: Proceedings of the Second International Conference on Advances in Information Systems**, páginas 405–414, London, UK, 2002. Springer-Verlag. ISBN 3-540-00009-7.

N. Nedjah e L. M. Mourelle. Parallel computation of modular exponentiation for fast cryptography. **International Journal of High Performance Systems Architecture**, 1(1):44–49, 2007.

J. M. Pollard. Factoring with cubic integers. manuscript, 1988.

C Pomerance. The quadratic sieve factoring algorithm. In: **Proc. of the EUROCRYPT 84 workshop on Advances in cryptology: theory and application of cryptographic techniques**, páginas 169–182, New York, NY, USA, 1985. Springer-Verlag New York, Inc. ISBN 0-387-16076-0.

Carl Pomerance, J. W. Smith, e Randy Tuler. A pipeline architecture for factoring large integers with the quadratic sieve algorithm.

SIAM J. Comput., 17:387–403, April 1988. ISSN 0097-5397. URL http://portal.acm.org/citation.cfm?id=45474.45486.

R. L. Rivest, A. Shamir, e L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. **Commun. ACM**, 21(2):120–126, 1978. ISSN 0001-0782.

RSA Labs. PKCS#1 v2.0 Amendment 1: Multi-Prime RSA, 2000.

Igor A. Semaev. An algorithm for evaluation of discrete logarithms in some non-prime finite fields. **Math. Comput.**, 67(224):1679–1689, 1998. ISSN 0025-5718.

Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, e Jack Dongarra. **MPI-The Complete Reference, Volume 1: The MPI Core**. MIT Press, Cambridge, MA, USA, 1998. ISBN 0262692155.

Apêndice A

Teste de Primalidade Miller-Rabin

O teste Miller-Rabin Miller (1975) é um algoritmo probabilístico para testar a primalidade de um inteiro ímpar n. Este teste está baseado nos seguintes fatos:

(1) Se n é um primo ímpar as únicas raízes quadradas de $1 \mod n$ são 1 e $-1(\equiv (n-1))$. Se n fosse composto e não fosse uma potência de um primo, então 1 possui outras raízes quadradas módulo n.

(2) Se n é um primo ímpar podemos escrever $n-1 = 2^s r$ onde r é ímpar. Seja $a \in \mathbb{Z}$ e $mdc(a,n) = 1$. Então $a^r \equiv 1 \mod n$ ou $a^{2^j r} \equiv -1 \mod n$ para algum j, $0 \leq j \leq (s-1)$.

Estas idéias motivas a seguinte definição Menezes et al. (1997):

Definição A.0.0.1 Seja n um inteiro ímpar e onde $n-1 = 2^s r$ onde r é um ímpar. E seja a definido no intervalo $1 \leq a \leq n-1$

i) Se $a^r \not\equiv 1 \mod n$ e $a^{2^j r} \not\equiv -1 \mod n$ para todo j, $0 \leq j \leq (s-1)$ então a é definido como "forte testemunho" da não primalidade de n.

ii) Se, no entanto, $a^r \equiv 1 \mod n$ ou $a^{2^j r} \equiv -1 \mod n$ para algum j, $0 \leq j \leq (s-1)$ então chamamos n de "forte pseudo-primo" para a base a. E o inteiro a é definido como "falso testemunho" para a primalidade de n.

O parâmetro de segurança t do algoritmo que veremos indica qual a probabilidade de um inteiro composto ímpar n ser declarado como primo. Neste caso, temos uma chance menor que $\left(\frac{1}{4}\right)^t$. Com estas informações o algoritmo 22 mostra o funcionamento do teste de primalidade devido a Miller-Rabin.

Algoritmo 22: Teste probabilístico de primalidade Miller-Rabin.

Entrada: Um inteiro ímpar n e um parâmetro de segurança t
Saída: A reposta para a pergunta: "n é ou não um primo?"

```
início
    Escreva n − 1 = 2^s r onde r é um ímpar;
    para i ← 1 até t faça
        Escolha um inteiro aleatório a, 2 ≤ a ≤ n − 2;
        y ← a^r mod n;
        se y ≠ 1 E y ≠ (n − 1) então
            j ← 1;
            enquanto j ≤ (s − 1) E y ≠ (n − 1) faça
                y ← y^2 mod n;
                se y = 1 então retorna "Composto";
                j ← j + 1;
            se y ≠ (n − 1) então retorna "Composto";
    retorna "Primo";
fim
```