Paralelização Eficiente para o Algoritmo Binário de Exponenciação Modular

Pedro Carlos da Silva Lara[1], **Fábio Borges de Oliveira**[1], **Renato Portugal**[1]

[1]Laboratório Nacional de Computação Científica – LNCC
Av. Getúlio Vargas, 333 - Quitandinha 25.651-075 - Petrópolis, RJ

{pcslara,borges,portugal}@lncc.br

Abstract. *Algorithms for modular exponentiation play an important role in asymmetric cryptography. The efficiency of RSA, for instance, depends on algorithms for modular exponentiation. This operation is the most expensive part in many methods of cryptography, for instance, in the Diffie-Hellman protocol. An efficient implementation of modular exponentiation has a strong impact on the performance of those methods. In this paper, a modification of the algorithm for binary modular exponentiation is proposed, which exploits a method of parallelization and reduces the computational complexity of the algorithm by a quadratic factor in the number of multiplications.*

Resumo. *Algoritmos de exponenciação modular têm um papel importante na criptografia assimétrica. O desempenho do RSA, por exemplo, depende de um algoritmo de exponenciação modular. Esta operação é a mais custosa em muitos métodos de criptografia, por exemplo, no protocolo Diffie-Hellman. Uma implementação eficiente da exponenciação modular tem forte impacto sobre estes métodos.Neste trabalho, é proposta uma modificação do algoritmo binário de exponenciação modular, que explora um método de paralelização e reduz a complexidade do algoritmo por um fator quadrático no número de multiplicações.*

1. Introdução

Whitfield Diffie e Martin E. Hellman [Diffie and 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 $k_b = g^b$. Agora ambos usam seu inteiro secreto para obter uma chave secreta compartilhada $k_{ab} = (k_a)^b = (k_b)^a = g^{ab}$. 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 and 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 and 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 and Mourelle 2007] discutem e comparam a paralelização de alguns dos principais algoritmos de exponenciação modular.

Este artigo propõe um novo método de paralelização baseado em uma modificação do algoritmo binário de exponenciação modular. Nesta nova versão, o número de multiplicações se reduz por um fator quadrático em média em relação ao algoritmo sequencial. O algoritmo de paralelização foi implementado em linguagem `C` afim de ser comparado com o algoritmo sequencial. Na seção 2, é feita uma breve revisão do algoritmo binário de exponenciação modular. Na seção 3, a ideia central deste artigo é descrita: a paralelização do algoritmo binário. Na seção 4, é feita a análise de complexidade. Na seção 5, os testes são mostrados através de gráficos. Na última seção, nossas conclusões serão apresentadas.

2. Algoritmo Binário de 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 1 processa os *bits* da esquerda para a direita. Analogamente, o algoritmo 2 processa os *bits* da direita para a esquerda.

Algoritmo 1: Algoritmo binário versão esquerda para direita.

Entrada: Inteiro $g \in \mathbb{Z}_p$ e $e = \sum_{i=0}^{j} 2^i b_i$ onde $b_i \in \{0, 1\}$ (representação em base binária).
Saída: $g^e \mod p$.

```
1 início
2     a ← 1;
3     para i = j até 0 faça
4         a ← a^2 mod p;
5         se b_i = 1 então
6             a ← a · g mod p;
7     retorna a;
8 fim
```

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

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

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

Algoritmo 2: Algoritmo binário versão direita para esquerda.

Entrada: Inteiro $g \in \mathbb{Z}_p$ e $e = \sum_{i=0}^{j} 2^i b_i$ onde $b_i \in \{0, 1\}$ (representação em base binária).
Saída: $g^e \mod p$.

```
1 início
2     a ← 1;
3     para i = 0 até j faça
4         se b_i = 1 então
5             a ← a · g mod p;
6         g ← g^2 mod p;
7     retorna a;
8 fim
```

Apesar de grande importância no desempenho final do algoritmo, não iremos discutir, neste trabalho, as operações de multiplicação, elevação ao quadrado e redução modular em $\mathbb{Z}_p$. O leitor poderá consultar [Menezes et al. 1997, Koç 1994, Bosselaers et al. 1994]. Para a aritmética de precisão múltipla ver [Knuth 1997]. A complexidade computacional de ambos algoritmos binários de exponenciação modular é $j+1$ elevações ao quadrado e $\frac{j+1}{2}$ multiplicações modulares em média, onde $j+1$ é o número de *bits* do expoente e.

3. Paralelização do Algoritmo Binário

Quando levamos em conta a representação em base binária do expoente $e = (b_j b_{j-1} \ldots b_1 b_0)_2$, podemos identificar a seguinte identidade

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

onde $e_1 = (b_r \ldots b_1 b_0)_2$, $e_2 = (b_j \ldots b_{r+1})_2$ e $r = \lceil (j-1)/2 \rceil$. Na verdade, a equação (1) segue do fato que:

$$e = 2^{r+1}e_2 + e_1. \tag{2}$$

Isto posto, podemos computar as parcelas $g^{2^{r+1}e_2}$ e g^{e_1} de (1) em separado sem dependência mútua. A equação (1) resume a ideia central da paralelização do algoritmo binário de exponenciação. A figura 1 exemplifica, computacionalmente, estes passos. O algoritmo 3 sistematiza estas ideias.

Quando $j+1$ (número de *bits* de e) for par, a "separação" do expoente e será igualmente dividida. e_1 e e_2 ficam, cada um, com $\frac{j+1}{2}$ *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 3) é mais custosa que na Região Paralela 1, é mais sensato, quando $j+1$ 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 bastante desejado. O algoritmo 3 atua com duas linhas paralelas de execução, no entanto, essa ideia pode ser mais geral - poderíamos ter $e_1, e_2, \ldots, e_n$ onde n é o número de vias paralelas de execução e proceder de forma análoga ao que foi mostrado no algoritmo 3 (veja o algoritmo 4).

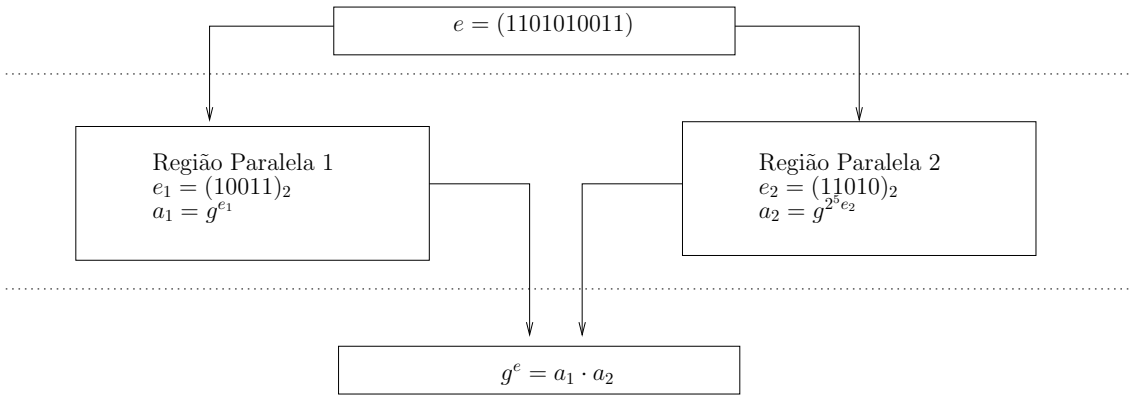

Figura 1. Exemplo com duas linhas paralelas.

Algoritmo 3: Paralelização do algoritmo binário.

Entrada: Inteiro $g \in \mathbb{Z}_p$ e $e = \sum_{i=0}^{j} 2^i b_i$ onde $b_i \in \{0, 1\}$ (representação em base binária).
Saída: $g^e \mod p$.

```
1  início
2      e1 ← (br ... b1b0)2;
3      e2 ← (bj ... br+1)2;
4      início
5          [Região Paralela 1]
6          a1 ← g^e1 mod p (algoritmo 1 ou 2);
7      fim
8      início
9          [Região Paralela 2]
10         a2 ← g^(2^(r+1));
11         a2 ← a2^e2 mod p (algoritmo 1 ou 2);
12     fim
13     retorna a1 · a2 mod p;
14 fim
```

No algoritmo 4, o tamanho do expoente e_i para $i = 1, \ldots, n$ também deve ser levado em conta. Nesse caso, o resto da divisão do número de *bits* de e por n de ser analisado.

4. Análise de Complexidade

Voltando ao exemplo numérico da figura 1, é fácil perceber que o lado direito (Região Paralela 2 do algoritmo 3) exige uma computação maior. Neste caso, quando computamos $a_2 = g^{2^{r+1}}$ na verdade estamos calculando $r + 1$ elevações ao quadrado e 1 multiplicação (veja algoritmo 1 ou 2). E, finalmente, quando fazemos $a_2 = a_2^{e_2}$ fazemos $r + 1$ elevações ao quadrado e $\frac{r+1}{2}$ multiplicações em média. Assim, a Região Paralela 2 do algoritmo 3 consome, aproximadamente, $2(r + 1)$ elevações ao quadrado e $\frac{r+1}{2}$ multiplicações. Se somarmos a multiplicação computada na linha 13 do algoritmo 3, temos um custo com-

Algoritmo 4: Paralelização com n linhas de execução.

Entrada: Inteiro $g \in \mathbb{Z}_p$ e $e = \sum_{i=0}^{j} 2^i b_i$ onde $b_i \in \{0, 1\}$ (representação em base binária).

Saída: $g^e \mod p$.

início
  $e_1 \leftarrow (b_{r_1} \ldots b_1 b_0)_2$;
  $e_2 \leftarrow (b_{r_2} \ldots b_{r_1+1})_2$;
  $\vdots$
  $e_n \leftarrow (b_j \ldots b_{r_{n-1}+1})_2$;
  início
    [Região Paralela 1]
    $a_1 \leftarrow g^{e_1} \mod p$;
  fim
  início
    [Região Paralela 2]
    $a_2 \leftarrow g^{2^{r_1}}$;
    $a_2 \leftarrow a_2^{e_2} \mod p$;
  fim
  $\vdots$
  início
    [Região Paralela n]
    $a_n \leftarrow g^{2^{r_{n-1}}}$;
    $a_n \leftarrow a_n^{e_n} \mod p$;
  fim
  retorna $a_1 \cdot a_2 \cdot \ldots \cdot a_n \mod p$;
fim

putacional de

$$2(r+1)E + \left(\frac{r+1}{2} + 1\right) M, \tag{3}$$

onde M é o tempo requerido para uma multiplicação e E é o tempo computacional para uma elevação ao quadrado. Essa análise vale quando $j+1$ é par, que é o pior caso. Já em função de j, a expressão (3) fica

$$(j+1)E + \left(\frac{j+1}{4} + 1\right) M. \tag{4}$$

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

$$(r+1)E + \frac{r+1}{2}M,$$

ou seja, faz $(r+1)E$ a menos que a Região Paralela 2. Se compararmos com a complexidade do algoritmo binário sequencial, que é de $(j+1)E + \left(\frac{j+1}{2}\right) M$, com a complexidade do algoritmo paralelo (veja expressão (4)), temos uma diferença de $\frac{j+1}{4}M$ aproximadamente.

Não é difícil generalizar estas ideias para n linhas paralelas. Para isso basta tomar $r = \frac{j-1}{n}$ e lembrar que, ao final do algoritmo (linha 21 do algoritmo 4), $n-1$ multiplicações serão executadas. Na verdade, o número de elevações ao quadrado continua sendo $(j+1)$, no entanto, o número de multiplicações fica $\frac{j+1}{2n}+n-1$. Generalizando, temos um custo computacional de:

$$(j+1)E + \left(\frac{j+1}{2n} + n - 1\right) M. \tag{5}$$

Assim, em todos os casos sempre fazemos $(j+1)E$. A diferença fica com o número de multiplicações. Desta forma, para fazer uma análise um pouco mais detalhada precisamos explorar a função $\eta(j,n)$ que retorna no número de multiplicações definida por

$$\eta(j,n) = \frac{j+1}{2n} + n - 1. \tag{6}$$

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

$$\frac{\partial \eta(j,n)}{\partial n} = -\frac{j+1}{2n^2} + 1. \tag{7}$$

Igualando (7) a zero e resolvendo a equação na variável n, temos a seguinte relação entre o número de linhas paralelas e o número de *bits* do expoente

$$n = \sqrt{\frac{j+1}{2}}. \tag{8}$$

Como a concavidade de $\eta(j,n)$ é positiva, segue que a equação (8) fornece um mínimo global. Este é o número ótimo de linhas paralelas n em função do número de *bits* $j+1$, que determina a segurança do criptossistema.

Tabela 1. Comparação entre as implementações

Algoritmo	Tamanho das entradas	Tempo médio de execução	σ
Paralelo 4	512	426.21	41.49
	1024	2127.82	129.31
	2048	12315.27	798.10
Paralelo 2	512	512.52	64.29
	1024	2641.66	228.63
	2048	15443.36	980.86
Sequencial	512	568.66	9.36
	1024	3107.93	122.26
	2048	18524.82	266.72

5. Metodologia e Testes de Desempenho

A implementação foi desenvolvida em linguagem `C` usando como ferramenta básica de paralelização a biblioteca OpenMP (Open Multi-Processing) [Eigenmann and Ayguade 2009]. A API (Application Programming Interface) do OpenMP oferece, de maneira eficaz, a paralelização usando memória física compartilhada. 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. Para os testes, utilizamos o processador `Intel(R) Quad Core(TM) Q9300` com `2.50GHz` de frequência e `3072 KB` de memória *cache*. Para comparar o desempenho, implementamos três algoritmos: o algoritmo binário que processa os *bits* do mais para o menos significativo (algoritmo 1), a sua paralelização (algoritmo 3) e finalmente, utilizamos uma versão do algoritmo 3 com 4 linhas paralelas. A tabela 5 compara os resultados de 1000 coletas sucessivas. O tamanho da entrada, em *bits*, foi de 512, 1024 e 2048, respectivamente. A última coluna desta tabela mostra o desvio padrão σ do tempo de execução em microssegundos.

Como era de se esperar, a diferença no tempo de execução é mais evidente a medida que duas variáveis crescem: número de *bits* da entrada e número de linhas paralelas de execução. O ganho em porcentagem com relação ao algoritmo binário sequencial é apresentado na figura 2.

O paralelo 4 apresenta um ganho de, aproximadamente, 33% em relação ao algoritmo sequencial para entradas de 2048 *bits*. Este ganho é razoável, pois estamos testando a paralelização para um número de processadores abaixo do ótimo. Lembre que a eficiência computacional de muitos algoritmos de criptografia assimétrica está diretamente ligada com o desempenho do algoritmo de exponenciação modular. A figura 3 exibe um gráfico comparativo das três implementações.

6. Conclusões e Trabalhos Futuros

Este artigo descreveu um método de paralelização para o cálculo da exponenciação modular. Explanamos o algoritmo paralelo proposto e ilustramos através de gráficos e tabelas o desempenho computacional do mesmo. Comparamos os resultados com o respectivo algoritmo sequencial. Como trabalhos futuros, pretendemos estender a ideia central que foi mostrada neste artigo para métodos que possuem melhor desempenho computacional

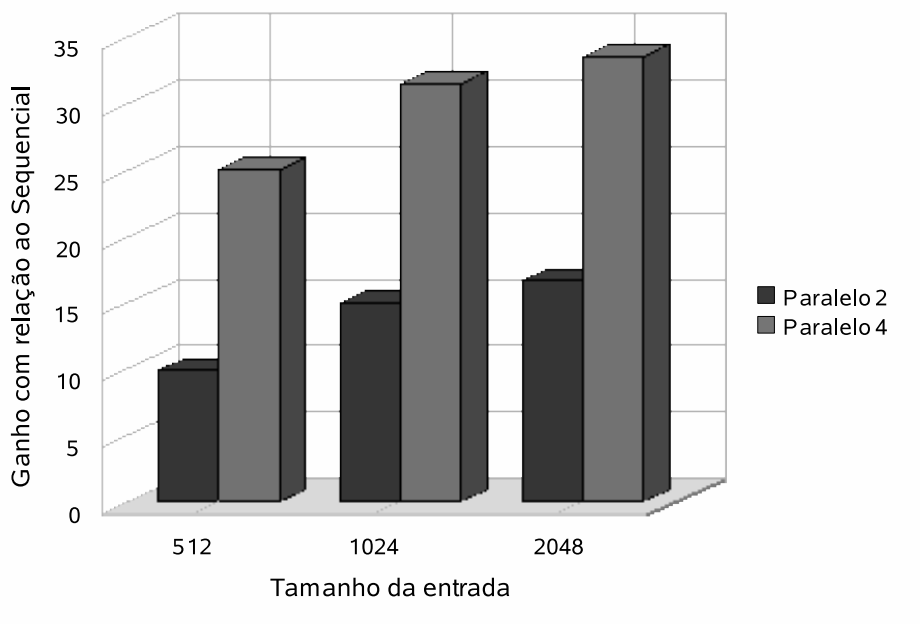

Figura 2. Ganho em porcentagem em relação ao algoritmo binário sequencial.

se comparado ao método binário de exponenciação modular, tais como: Método k-ário e Janela Deslizante. 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 algoritmos 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.

7. Agradecimentos

Gostaríamos de agradecer aos revisores pelas importantes sugestões e o suporte financeiro oferecido pelo CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico).

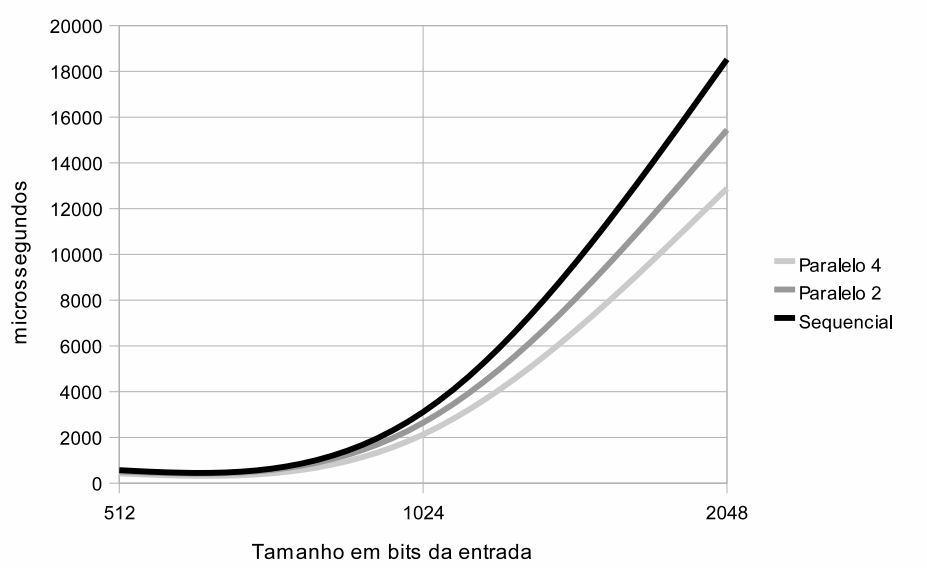

Figura 3. Comparativo de desempenho entre implementações.

Referências

Bosselaers, A., Govaerts, R., and Vandewalle, J. (1994). Comparison of three modular reduction functions. In *In Advances in Cryptology-CRYPTO'93, LNCS 773*, pages 175–186. Springer-Verlag.

Brickell, E. F. (1990). A survey of hardware implementation of RSA. In *CRYPTO '89: Proceedings of the 9th Annual International Cryptology Conference on Advances in Cryptology*, pages 368–370, London, UK. Springer-Verlag.

Brickell, E. F., Gordon, D. M., Mccurley, K. S., and Wilson, D. B. (1992). Fast exponentiation with precomputation. In *Advances in Cryptology – Proceedings of CRYPTO'92*, volume 658, pages 200–207. Springer-Verlag.

Brickell, E. F., Gordon, D. M., Mccurley, K. S., and Wilson, D. B. (1995). Fast exponentiation with precomputation: Algorithms and lower bounds. Preprint `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.5.606`.

Diffie, W. and Hellman, M. E. (1976). New directions in cryptography. *IEEE Trans. Information Theory*, IT-22(6):644–654.

Eigenmann, R. and Ayguade, E. (2009). Special Issue: OpenMP Introduction. *International Journal of Parallel Programming*, 37(3):247–249.

Granlund, T. (2002). GNU multiple precision arithmetic library 4.1.2. `http://swox.com/gmp/`.

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

Koç, C. K. (1994). High-speed RSA implementation. Technical Report, RSA Laboratories.

Lim, C. H. and Lee, P. J. (1994). More flexible exponentiation with precomputation. In *Advances in Cryptology – CRYPTO'94*, pages 95–107.

Menezes, A. J., Oorschot, P. C. V., Vanstone, S. A., and Rivest, R. L. (1997). Handbook of applied cryptography.

Nedjah, N. and Mourelle, L. M. (2002). Efficient parallel modular exponentiation algorithm. In *ADVIS '02: Proceedings of the Second International Conference on Advances in Information Systems*, pages 405–414, London, UK. Springer-Verlag.

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

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

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