Journal of Computer and System Sciences 78 (2012) 575–582

Contents lists available at ScienceDirect

Journal of Computer and System Sciences

www.elsevier.com/locate/jcss

Parallel modular exponentiation using load balancing without precomputation

Pedro Lara [a,*], Fábio Borges [a], Renato Portugal [a], Nadia Nedjah [b]

[a] National Laboratory for Scientific Computing, Petrópolis, RJ, 25651-075, Brazil
[b] Faculty of Engineering, State University of Rio de Janeiro, Rio de Janeiro, RJ, 20550-900, Brazil

ARTICLE INFO

Article history:
Received 19 October 2010
Received in revised form 25 July 2011
Accepted 28 July 2011
Available online 11 August 2011

Keywords:
Modular exponentiation
Load balancing
Cryptography

ABSTRACT

The modular exponentiation operation of the current algorithms for asymmetric cryptography is the most expensive part in terms of computational cost. The RSA algorithm, for example, uses the modular exponentiation algorithm in encryption and decryption procedure. Thus, the overall performance of those asymmetric cryptosystems depends heavily on the performance of the specific algorithm used for modular exponentiation. This work proposes new parallel algorithms to perform this arithmetical operation and determines the optimal number of processors that yields the greatest speedup. The optimal number is obtained by balancing the processing load evenly among the processors. Practical implementations are also performed to evaluate the theoretical proposals.

© 2011 Elsevier Inc. All rights reserved.

1. Introduction

RSA (Rivest, Shamir and Adleman) [1] for asymmetric cryptography and Diffie–Hellman protocol [2] for key exchange are widely used in trading and secure communication. It is impossible to overstress the importance of those methods in practical applications. Their security depends upon the difficulty in factoring large integers and the main arithmetical operation used in the whole process is the integer modular exponentiation. Then, the security and usefulness of those methods depend upon efficient algorithms for modular exponentiation.

Many kinds of implementations were suggested to enhance the efficiency of modular exponentiation. Brickell [3] has discussed implementation of RSA in hardware. Brickell, Gordon, McCurley and Wilson (BGMW) have used precomputation to speedup the exponentiation in Ref. [4] and have proposed two parallelization methods using precomputation [5]. Lim and Lee have also discussed parallelization using the BGMW method [6]. These techniques are specially useful in cryptographic methods that use fixed base, such as Diffie–Hellman protocol.

Many research works have been focused on a fixed base [6–9]. In [10], J. Von Zur Gathen describes an exponentiation algorithm in $GF(2^m)$ without pre-computing. That algorithm was improved by Mun-Kyu Lee as reported in [11], where the fact that squaring operation in $GF(2^m)$ is linear using a normal base was exploited.

Parallel algorithms for modular exponentiation useful in RSA were discussed by Nedjah and Mourelle [12–14]. They have compared the main parallelization methods. RSA using multiple primes [15] via Chinese remainder theorem provides an efficient parallelization method.

Recently, parallel algorithms are playing an important role in keeping up the exponential growth predicted by Moore's law. Not only multi-core processors but also powerful graphic cards are becoming more available. Then, flexibility in choosing parallelization methods is welcome in that context.

* Corresponding author.
E-mail addresses: pcslara@lncc.br (P. Lara), borges@lncc.br (F. Borges), portugal@lncc.br (R. Portugal), nadia@eng.uerj.br (N. Nedjah).

0022-0000/$ – see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jcss.2011.07.002

This work proposes a new method of load balancing based on the parallelization algorithm for modular exponentiation discussed in [16]. We present an optimization method to obtain the best number of processors the distribute the processing load evenly. Practical implementations also presented here confirm our analysis. The load balancing has helped to reduce significantly the number of processors for exponents of 1024 bit used nowadays in cryptography. The execution time has also decreased.

The paper is organized as follows. In Section 2 we briefly review the square and multiply method. In Section 3 we review the main results obtained in [16]. In Section 4 we present theoretical methods to obtain the optimal partition points that allows for a load balancing. In Section 5 we perform numerical experiments to check the theoretical proposals. In the last section we draw our conclusions.

2. Binary modular exponentiation

The square and multiply method for modular exponentiation is around two thousand years old [17]. The central idea to calculate $g^e \mod p$ is to use the binary representation of the exponent e. The arithmetical properties ensures that

$$g^e \mod p = \left(\cdots\left(\left(g^{b_n}\right)^2 \cdot g^{b_{n-1}}\right)^2 \cdots\right)^2 \cdot g^{b_1} \mod p, \tag{1}$$

where $e = \sum_{i=1}^{n} 2^i b_i$ and $b_i \in \{0, 1\}$. Starting the calculation by the innermost parenthesis, that is (g^{b_n}), one can see that the number of squaring and multiplication is at most equal to the number of bits of the exponent.

Algorithm 1: Binary modular exponentiation.

Input: Integer $g \in \mathbb{Z}_p$ and $e = \sum_{i=1}^{n} 2^i b_i$ where $b_i \in \{0, 1\}$.
Output: $g^e \mod p$.

```
1 a ← 1
2 for i = n to 1 do
3 |   a ← a^2 mod p
4 |   if b_i = 1 then
5 |   |   a ← a · g mod p
6 return a
```

Algorithm 1 implements the modular exponentiation using Eq. (1). The bits of the exponent are used from left to right. There is an alternative version that uses the bits from right to left. The exact number of modular squaring is n and the average number of modular multiplications is $\frac{n}{2}$, where n is the number of bits of the exponent e.

The details of multiplying, squaring and performing modular reduction in $\mathbb{Z}_p$ are important to the total cost, but we do not discuss those issues here. Refs. [18–20] provide useful material about them and Ref. [17] about multi-precision arithmetics.

3. Parallelization

Let us start describing how to use two processors to speed up the calculation of the modular exponentiation. Using the fact that the exponent $e = (b_n b_{n-1} \cdots b_1 b_1)_2$ can be split up as

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

where $e_1 = (b_r \cdots b_1 b_1)_2$, $e_2 = (b_n \cdots b_{r+1})_2$ and $r = \lceil n/2 \rceil$, we can factorize

$$g^e = g^{2^r e_2} \cdot g^{e_1}. \tag{3}$$

The factors $g^{2^r e_2}$ and g^{e_1} have no mutual dependence. They can be computed in parallel using Algorithm 1 on each factor. The results obtained by the included processors must be multiplied at the end, and thus yielding the final result. The number of bits of e_1 and e_2 are not necessarily equal, since the number of factors $k = 2$ may not divide n. Even when they have exactly the same number of bits, the load is not balanced, because clearly $g^{2^r e_2}$ has more squares than g^{e_1}.

The same ideas used for 2 processors can be generalized to k processors assuming that e is partitioned in parts of the same size. In the following description, let us suppose that the number of factors k divides n and let $r_1 = 0, r_2 = \frac{n}{k}, r_3 = \frac{2n}{k}, \ldots, r_k = \frac{(k-1)n}{k}$. So, now Eq. (3) can be re-written as:

$$g^e = g^{2^{r_k} e_k} \cdots g^{2^{r_2} e_2} \cdot g^{2^{r_1} e_1},$$

where $e_1, e_2, \ldots, e_k$ are defined as in Algorithm 2, which describes the parallelization scheme with k processors.

If g has a fixed value (fixed-base) we could pre-compute $g^{2^{r_i}}$ for $i \in \{1, 2, \ldots, k\}$ and reduce substantially the computational time in parallelization, making the sublinear speedup. However, not all applications use a fixed base. For instance, in the RSA cryptosystem, the base is different in each encryption/decryption process. The major overhead imposed by this

parallelization is the computation of squaring, in this case, g^{2^k}. So far, there is no parallel algorithm for this process. This computation have a strong sequential dependency. Therefore, many research work was dedicated to the case where the base g is known *a priori*.

Algorithm 2: Parallelization using k processors.

Input: Integer $g \in \mathbb{Z}_p$ and $e = \sum_{i=1}^{n} 2^i b_n$ where $b_i \in \{0, 1\}$.
Output: $g^e \bmod p$.

```
1  e_1 ← (b_{r_2} ··· b_1 b_1)_2
2  e_2 ← (b_{r_3} ··· b_{r_2+1})_2
3  ⋮
4  e_k ← (b_n ··· b_{r_k+1})_2
5  begin
6  |  [Factor #1]
7  |  a_1 ← g^{2^0} mod p
8  |  a_1 ← a_1^{e_1} mod p
9  end
10 begin
11 |  [Factor #2]
12 |  a_2 ← g^{2^{r_2}} mod p
13 |  a_2 ← a_2^{e_2} mod p
14 end
15 ⋮
16 begin
17 |  [Factor #k]
18 |  a_k ← g^{2^{r_k}} mod p
19 |  a_k ← a_k^{e_k} mod p
20 end
21 return a_1 · a_2 · ····· · a_k mod p
```

3.1. Cost

In this case, the factor #k in Algorithm 2 has the greatest load. The computation of $g^{2^{r_{k-1}}}$ (line 2) requires $n - \frac{n}{k}$ squares and $a_k^{e_k}$ (line 2) requires $\frac{n}{k}$ squares and $\frac{n}{2k}$ multiplications in average. Adding the cost of all factors including the cost of line 2, which is $k - 1$ multiplications, we obtain

$$nS + \left(\frac{n}{2k} + k - 1\right)M, \tag{4}$$

where S and M are the cost of one square and one multiplication respectively. Note that the number of squares is the same as before parallelizing but the number of multiplications gets reduced. The reduction is a direct consequence of splitting up exponent e into smaller parts.

Now, let us calculate the optimal number of factors. Let $\eta(n, k)$ be the number of multiplications, that is

$$\eta(n, k) = \frac{n}{2k} + k - 1. \tag{5}$$

Keeping n constant, the optimal value of k is the one that satisfies the equation $\frac{\partial \eta(n,k)}{\partial k} = 0$. The solution is given by

$$k_0 = \sqrt{\frac{n}{2}}. \tag{6}$$

The second derivative $\frac{\partial^2 \eta(n,k)}{\partial k^2}$ is positive, then k_0 is a global minimum. This value is the optimal number of processors depending on the number of bits of the exponent. Replacing k in Eq. (4) by k_0, we obtain

$$nS + (\sqrt{2n} - 1)M,$$

which shows that the number of multiplications is reduced by a quadratic factor in average for large n when we compare with Eq. (4) taking $k = 1$. The ideal number of processors is approximately square root of half of the number of bits of the exponent. For RSA-1024, 23 processors would be necessary.

4. Load balancing

Algorithm 2 does not distribute the workload evenly across the processors when the partition is uniform. Let us suppose that Algorithm 2 has only two parallel factors. Let us call T_1 the total cost to process Factor #1. It is given by

$$T_1 = \frac{n}{2}S + \frac{n}{4}M.$$

The total cost of Factor #2 has extra squarings and is given by

$$T_2 = nS + \frac{n}{4}M.$$

The computational cost of Factor #2 is greater than Factor #1. Then

$$T_2 - T_1 = \frac{n}{2}S.$$

Instead of considering partitions of the same size, let us find the point p for partitioning such that we have $T_1 = T_2$. The computational costs to calculate the factors are now given by

$$T_1 = pS + \frac{p}{2}M,$$
$$T_2 = nS + \left(\frac{n-p}{2}\right)M.$$

Without loss of generality, let us assume that $S = 1$, that is, one squaring takes one time unit and let us define $\varphi = \frac{M}{S}$. Equation $T_1 = T_2$ implies that

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

The general case with n processors requires the identification of a sequence of partition points $p_1, p_2, \cdots, p_k$ such that $p_k = n$ and $T_1 = T_2 = T_3 = \cdots = T_k$, that is, the sequence that allows for an even distribution of the workload among the available k processors. The computational cost of each factor is

$$T_1 = p_1 + \varphi\left(\frac{p_1}{2}\right) = p_1\left(1+\frac{\varphi}{2}\right),$$
$$T_2 = p_2 + \varphi\left(\frac{p_2-p_1}{2}\right) = p_2\left(1+\frac{\varphi}{2}\right) - \frac{\varphi}{2}p_1,$$
$$T_3 = p_3\left(1+\frac{\varphi}{2}\right) - \frac{\varphi}{2}p_2,$$
$$T_4 = p_4\left(1+\frac{\varphi}{2}\right) - \frac{\varphi}{2}p_3,$$
$$\vdots$$
$$T_i = p_i\left(1+\frac{\varphi}{2}\right) - \frac{\varphi}{2}p_{i-1},$$
$$\vdots$$
$$T_k = p_k\left(1+\frac{\varphi}{2}\right) - \frac{\varphi}{2}p_{k-1}.$$

Let us concentrate in the generic term $T_i = T_{i-1}$ for $i > 1$, which implies

$$p_i = (\lambda+1)p_{i-1} - \lambda p_{i-2},$$

where

$$\lambda = \frac{\varphi}{2+\varphi}. \tag{7}$$

This is a linear recurrence equation with the following initial conditions

$$p_k = n, \tag{8}$$

$$p_1 = (\lambda + 1)^{-1} p_2. \tag{9}$$

The solution of a linear recurrence equation has the form

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

where x_1 and x_2 are the roots of the characteristic polynomial

$$\rho(x) = x^2 - (\lambda + 1)x + \lambda.$$

Then $x_1 = 1$ and $x_2 = \lambda$. Using the initial conditions (8) and (9) and the general form of the solution p_i, we obtain the following system of equations for constants α and β

$$\begin{cases} \alpha + \beta\lambda^k = n, \\ \alpha + \beta\lambda = (\lambda + 1)^{-1}\left(\alpha + \beta\lambda^2\right), \end{cases} \tag{10}$$

the solution of which is given by

$$\alpha = \frac{n}{1 - \lambda^k}, \tag{11}$$

$$\beta = \frac{n}{\lambda^k - 1}. \tag{12}$$

Then

$$p_i = \alpha\left(1 - \lambda^i\right). \tag{13}$$

That solution is acceptable only if there is an increasing sequence of integer partition points $p_1 < p_2 < \cdots < p_k$. This property can be checked by obtaining the derivative of p_i with respect to i. Considering k and n fixed, we get

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

Considering that $n > 2$ and $\log\lambda < 0$, it follows that $\frac{\partial p_i}{\partial i} > 0$, and hence, the sequence of partition points given by (13) is always increasing.

Note that the values of all partition points will be rounded to the nearest integers. Two neighboring points p_i and p_{i+1} can be rounded to the same value, if their difference is smaller than 1/2. In order to overcome this problem, we have to add the constraint

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

for all i. Using the fact that the first derivative $\frac{\partial p_i}{\partial i}$ is positive and the second derivative $\frac{\partial^2 p_i}{\partial i^2}$ is negative for all i, we conclude that the sequence of partition points is monotonic and the intervals $p_{i+1} - p_i$ are decreasing when i increases. So, the only difference we have to check is the last one $p_k - p_{k-1} > \frac{1}{2}$.

Using Eqs. (8) and (13) we obtain

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

Using Eq. (11), the constraint $p_k - p_{k-1} > \frac{1}{2}$ is expressed by

$$n > \frac{1}{2\lambda^{k-1}}\left(\frac{1 - \lambda^k}{1 - \lambda}\right). \tag{14}$$

Let us determine the optimal number of processors in terms of the length n of exponent e. Using Eq. (14) twice with k and $k + 1$, we obtain

$$\frac{1 - \lambda^k}{2\lambda^{k-1}(1 - \lambda)} < n < \frac{1 - \lambda^{k+1}}{2\lambda^k(1 - \lambda)}. \tag{15}$$

After isolating variable k, we obtain

$$\gamma < k < \gamma + 1, \tag{16}$$

where

$$\gamma = \log_\lambda \frac{1}{1 + 2n(1 - \lambda)}. \tag{17}$$

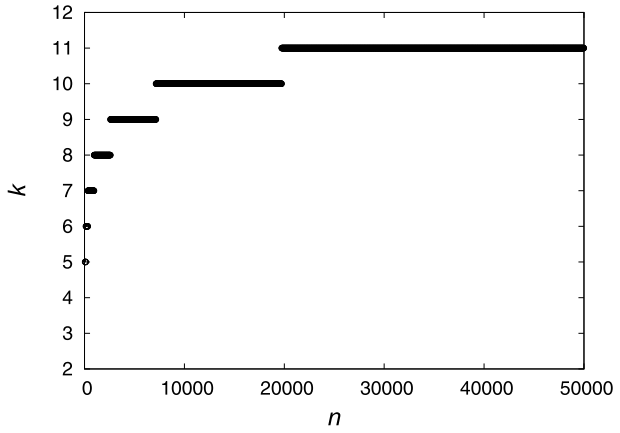

Fig. 1. The optimal number of processors k as function of the length n of the exponent.

To be sure that k is the correct integer number in the interval (16), we have to take

$$k = \left[\gamma + \frac{1}{2}\right], \tag{18}$$

where the notation [ ] denotes the nearest integer. Eq. (18) yields the optimal number of processors for a given n and φ. Note that the optimal number of processors has a logarithmic dependence on n, which is clearly displayed in Fig. 1.

Let us consider a practical example taking exponent with $n = 1024$ bits. A reasonable value of φ, the ratio between the time to perform one modular multiplication and the time to perform one squaring, is 1.14, as we will discuss in Section 5. Eq. (18) yields $k = 8$ as the best number of processors. In fact, Eq. (15) reduces to

$$944 < n < 2600$$

for that value of k, confirming our results. The partition points that produce the best load balancing are

$$p_1 = 652.42623896,$$
$$p_2 = 889.294363933,$$
$$p_3 = 975.291071726,$$
$$p_4 = 1006.5128064,$$
$$p_5 = 1017.84808587,$$
$$p_6 = 1021.96344211,$$
$$p_7 = 1023.45755234,$$
$$p_8 = 1024.0.$$

If instead we have taken erroneously $k = 9$, the partition points would be

$$p_1 = 652.300785978,$$
$$p_2 = 889.123364326,$$
$$p_3 = 975.103536083,$$
$$p_4 = 1006.31926723,$$
$$p_5 = 1017.65236707,$$
$$p_6 = 1021.76693199,$$
$$p_7 = 1023.26075492,$$
$$p_8 = 1023.80309827,$$
$$p_9 = 1024.0.$$

After rounding the last two points we would obtain $[p_8] = [p_9]$ and the final sequence would not be monotonic.

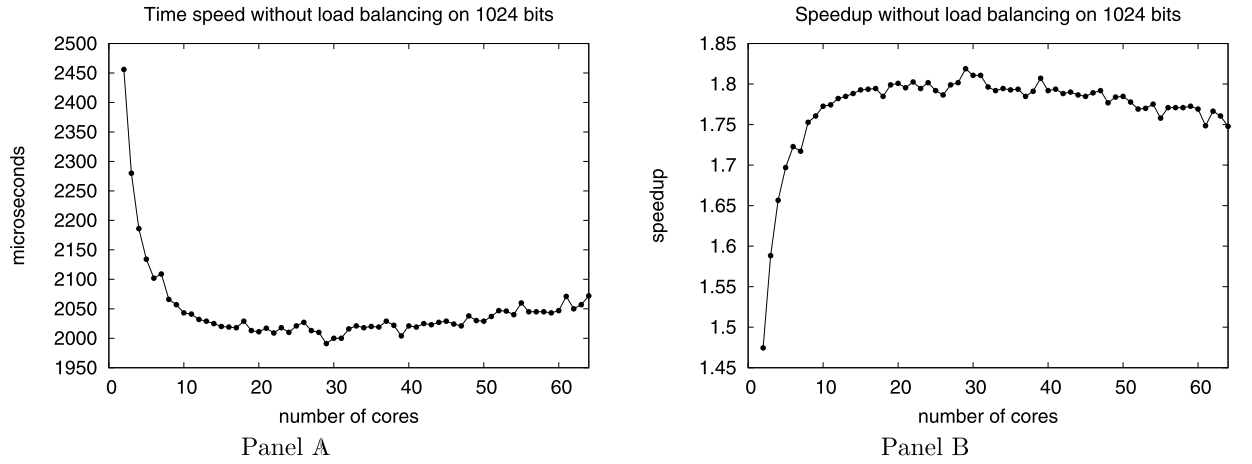

Fig. 2. Execution time (Panel A) and speedup (Panel B) up to 64 cores with no load balancing using exponents of 1024 bits.

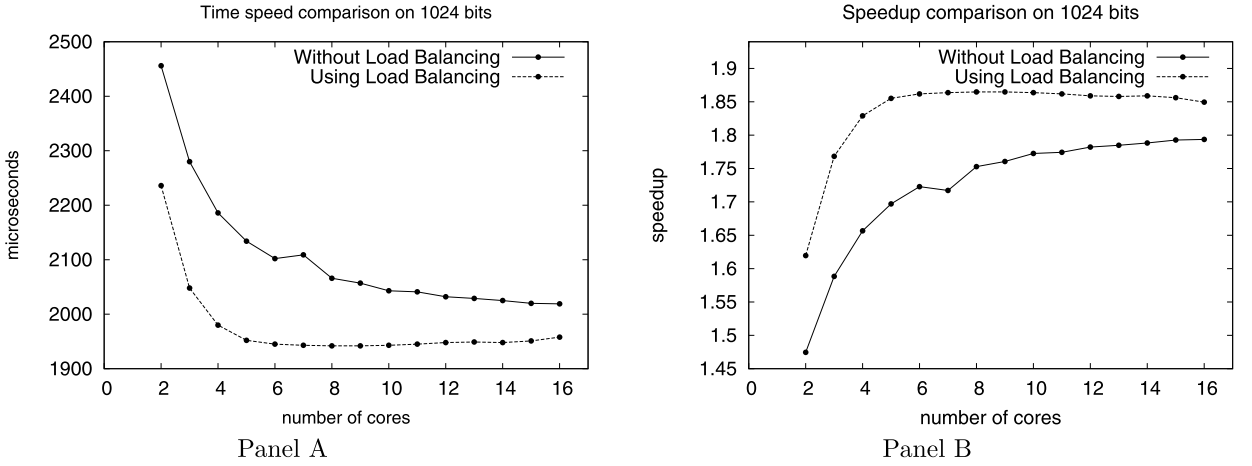

Fig. 3. Execution time (Panel A) and speedup (Panel B) up to 16 cores with load balancing using exponents with 1024 bits. The graphs with no load balancing are also depicted for comparison.

5. Implementations

The implementations using `C` Programming language of the algorithms proposed in this work use the OpenMPI parallelization library as a basic tool, which implements the standard MPI (Message Passing Interface) [21]. The API (Application Programming Interface) MPI allows the parallelization using distributed physical memory. For multiple precision arithmetic we used library GMP (GNU Multiple Precision) [22]. We used 8 nodes of a `Sun Blade x6250`. Each node has two processors `Intel Xeon E5440 Quad Core 3GHz x86 64` (64 *cores*) with 16 GB of memory physically interconnected by an *InfiniBand* bus. The operating system is CentOS Linux version 4.1.2 with kernel version 2.6.18. For all performance testing we used balanced exponents in binary representation. The execution time and the speedup up to 64 cores without using of load balancing are shown in Fig. 2. The time unit used is microsecond.

The graphs of Fig. 2 display oscillations specially when the number of cores is greater than 25 processors. If we disregard the oscillations, the curve of Panel A of Fig. 2 has a minimum for around 23 processors and the speedup in this case is around 1.8. The result is in agreement with the theoretical calculations of Section 3.

Let us now assess the impact of the load balancing technique. Fig. 3 displays the execution time and the speedup up to 16 cores of Algorithm 2 with load balancing. The graphs with no load balancing are also depicted for comparison. The execution time curve has a minimum for around 8 processors and the speedup in this case is around 1.86. For the platform that we are using, we have obtained the average value of $\varphi = 1.14$ for the ratio between the time to perform one modular multiplication and the time to perform one squaring. We have made the average over 100 samples. The ideal number of processors is in agreement with the theoretical calculations of Section 4.

The parallel computation of modular powers is more efficient when load balancing is applied. The best feature of the method that uses load balancing is the reduction of the number of processors. In the experiments with exponents of 1024 bit, the algorithm with load balancing needs 8 processors instead of 23 processors. The results are in agreement with

Table 1
Statistics using 100 samples comparing the timings for both techniques with the sequential calculation.

	Sequential	No balancing	Balancing
Arithmetic average	3621.38	1999.39	1961.45
Median	3613	1991	1942
Standard deviation	77.968	251.784	157.016
Number of processors	1	23	8

the theoretical analysis. Table 1 compares the execution times obtained using the techniques proposed in this work, *i.e.* with and without load balancing, to that yield by the sequential execution. The time unit is microsecond. As expected, they are faster than the sequential calculation. The running time is about 55% smaller. When we compare the parallelization schemes, the method that uses load balancing is better in all points. It uses less processors and the standard deviation is significantly smaller than the one obtained without load balancing (see Table 1).

6. Conclusions

This work has proposed a load balancing technique to optimize parallelization algorithms for the modular exponentiation operation. The parallelization scheme using uniform partition points is not the most efficient, because it does not use the available parallel resources with maximum load. The optimal number of processors depends on the number of bits of the exponent. For 1024 bits, the optimal number is 23 processors.

We have presented a theoretical analysis to obtain the optimal sequence of partition points that balance the workload of the processors. The optimal number of processors depends again on the number of bits of the exponent. For example, for 1024-bit exponents and the taking the coefficient $\varphi = 1.14$, we have obtained 8 processors for the optimal workload distribution. If we use more than 8 processors, there are redundant points in the partition. The high-load servers, for example *https* servers, may determine dynamically the optimal number of processors and the coefficient φ needed in the parallelization scheme.

The load balancing has advantages in two directions. The number of processors decreases and the speedup increases when compared to the uniform parallelization scheme. For exponents with 1024 bits, the optimal number of processors reduces from around 23 to 8, while the speedup increases from around 1.8 to around 1.86. A detailed comparison between the parallelization schemes is summarized in Table 1.

The practical implementations have confirmed our theoretical analysis.

References

[1] R.L. Rivest, A. Shamir, L. Adleman, A method for obtaining digital signatures and public-key cryptosystems, Commun. ACM 21 (1978) 120–126.
[2] W. Diffie, M.E. Hellman, New directions in cryptography, IEEE Trans. Inform. Theory IT-22 (1976) 644–654.
[3] E.F. Brickell, A survey of hardware implementation of RSA, in: CRYPTO '89: Proceedings of the 9th Annual International Cryptology Conference on Advances in Cryptology, Springer-Verlag, London, UK, 1990, pp. 368–370.
[4] E.F. Brickell, D.M. Gordon, K.S. McCurley, D.B. Wilson, Fast exponentiation with precomputation, in: Advances in Cryptology – Proceedings of CRYPTO'92, vol. 658, Springer-Verlag, 1992, pp. 200–207.
[5] E.F. Brickell, D.M. Gordon, K.S. McCurley, D.B. Wilson, Fast exponentiation with precomputation: Algorithms and lower bounds, preprint, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.5.606, 1995.
[6] C.H. Lim, P.J. Lee, More flexible exponentiation with precomputation, in: Advances in Cryptology – CRYPTO'94, pp. 95–107.
[7] J. von zur Gathen, Computing powers in parallel, SIAM J. Comput. 16 (1987) 930–945.
[8] J. von zur Gathen, Parallel algorithms for algebraic problems, in: STOC '83: Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing, ACM, New York, NY, USA, 1983, pp. 17–23.
[9] J.P. Sorenson, A sublinear-time parallel algorithm for integer modular exponentiation, 1999.
[10] J. von zur Gathen, Processor-efficient exponentiation in finite fields, Inform. Process. Lett. 41 (1992) 81–86.
[11] M.-K. Lee, Y. Kim, K. Park, Y. Cho, Efficient parallel exponentiation in $gf(2n)$ using normal basis representations, in: SPAA '01: Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures, ACM, New York, NY, USA, 2001, pp. 179–188.
[12] N. Nedjah, L.M. Mourelle, Efficient parallel modular exponentiation algorithm, in: ADVIS '02: Proceedings of the Second International Conference on Advances in Information Systems, Springer-Verlag, London, UK, 2002, pp. 405–414.
[13] N. Nedjah, L.M. Mourelle, Parallel computation of modular exponentiation for fast cryptography, Int. J. High Perform. Syst. Archit. 1 (2007) 44–49.
[14] N. Nedjah, L.M. Mourelle, High-performance hardware of the sliding-window method for parallel computation of modular exponentiations, Int. J. Parallel Program. 37 (2009) 537–555.
[15] RSA Labs, PKCS#1 v2.0 Amendment 1: Multi-Prime RSA, 2000.
[16] P.C.S. Lara, F. Borges, R. Portugal, Paralelização eficiente para o algoritmo binário de exponenciação modular, in: Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais, Campinas, SP.
[17] D.E. Knuth, Art of Computer Programming, vol. 2: Seminumerical Algorithms, 3rd edition, Addison–Wesley Professional, 1997.
[18] A.J. Menezes, P.C.V. Oorschot, S.A. Vanstone, R.L. Rivest, Handbook of Applied Cryptography, 1997.
[19] C.K. Koç, High-speed RSA implementation, Technical Report, RSA Laboratories, 1994.
[20] A. Bosselaers, R. Govaerts, J. Vandewalle, Comparison of three modular reduction functions, in: Advances in Cryptology-CRYPTO'93, in: Lecture Notes in Comput. Sci., vol. 773, Springer-Verlag, 1994, pp. 175–186.
[21] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra, MPI—The Complete Reference, vol. 1: The MPI Core, MIT Press, Cambridge, MA, USA, 1998.
[22] T. Granlund, GNU Multiple Precision Arithmetic Library 4.1.2, http://swox.com/gmp/, 2002.