Highly-Parallel Montgomery Multiplication for Multi-core General-Purpose Microprocessors

Selçuk Baktır[1] and Erkay Savaş[2]

[1] Bahçeşehir University
Department of Computer Engineering
Istanbul, Turkey
selcuk.baktir@bahcesehir.edu.tr
[2] SabancıUniversity
Faculty of Engineering & Natural Sciences
Istanbul, Turkey
erkays@sabanciuniv.edu

Abstract. Popular public key algorithms such as RSA and Diffie-Hellman key exchange, and more advanced cryptographic schemes such as Paillier's and Damgård-Jurik's algorithms (with applications in private information retrieval), require efficient modular multiplication with large integers of size at least **1024** bits. Montgomery multiplication algorithm has proven successful for modular multiplication of large integers. While general purpose multi-core processors have become the mainstream on desktop as well as portable computers, utilization of their computing resources have been largely overlooked when it comes to performing computationally intensive cryptographic operations. In this work, we propose a new parallel Montgomery multiplication algorithm which exhibits up to **39%** better performance than the known best serial Montgomery multiplication variant for the bit-lengths of **2048** or larger. Furthermore, for bit-lengths of **4096** or larger, the proposed algorithm exhibits better performance utilizing multiple cores available. It achieves speedups of up to **81%**, **3.37** times and **4.87** times for the used general-purpose microprocessors with **2**, **4** and **6** cores, respectively. To our knowledge, this is the first work that shows with actual implementation results that Montgomery multiplication can be practically parallelized on general-purpose multi-core processors.

Key Words: Montgomery multiplication, RSA, multi-core architectures, general-purpose microprocessors, parallel algorithms.

1 Introduction & Motivation

Many public key cryptosystems such as RSA, Diffie-Hellman, elliptic curve cryptography and recently pairing-based cryptography utilize multiplication as the most important operation which dominates the execution time. Therefore, the efficiency of multiplication operation determines the practicality and in some cases the feasibility of cryptographic applications.

Developing faster multiplication algorithms for larger numbers becomes the focal point of many research activities due to the ever increasing need for higher security levels. Emergence of multi-core processors on common desktop, notebook and server computers with no additional cost proclaim both the research opportunity and motivation for developing parallel algorithms for cryptographic applications.

Paillier encryption scheme [12], based on a setting similar to RSA, provides one of the most efficient and practical homomorphic encryption algorithms. Due to the fact that Paillier encryption scheme leads to message expansion after encryption Damgård and Jurik [3] generalize the algorithm for applications that require multiple encryption such as computationally-private information retrieval (CPIR) [10] and multi-hop homomorphic encryption scheme that encrypts already encrypted messages introduced in the scenario given in [6]. Especially in CPIR [10], for instance, the binary tree which aims to privately extract one data item out of a total of 256 eventually leads to a modular multiplication where the modulus size is 8192-bit for 80-bit security. For 128-bit security level, we have to perform multiplication operations with numbers as large as 24576-bit[3] for the same application.

Multi-core processors can be effectively put into use in executing parallelized multiplication operations of large numbers for accelerating aforementioned cryptographic applications. So far, the research on the subject has been focused on multi-core architectures [15], specifically built for multiplication operations of moderate size such as 1024 or 2048 bits, since inter-core communication dominates the overall computation in general-purpose multi-core processors for these bit lengths. However, we find out that this tendency starts changing for bit sizes of 2048-bit and higher if an efficient parallel multiplication algorithm is used.

Our Main Contributions:

- We present for the first time a practical parallel Montgomery multiplication algorithm [11] for general-purpose multi-core processors and present efficient implementation results.
- As in the case of single-core or hardware implementations, we show that the Montgomery multiplication algorithm turns out to render very efficient implementations on general-purpose multi-core processors due to its inherent parallelism. Our Montgomery multiplication algorithm

[3] Recommendation for Key Management, Special Publication 800-57 Part 1 Rev. 3, NIST, 05/2011.

demonstrates up to 39% better timing performance than the most common Montgomery multiplication algorithm in single-core software implementations for operand sizes of 2048-bit and larger.
– We analyze the timing performance of our algorithm on multi-core implementations on general-purpose multi-core processors. Using multi-core processors with 2, 4 and 6 cores, for operand sizes of 4096-bit and larger, we obtained speedups of up to 81%, 3.37 times and 4.87 times, respectively, compared with the most commonly used Montgomery multiplication algorithm.

2 Mathematical Background

2.1 Montgomery Multiplication

In many cryptographic algorithms, a chain of multiplication operations need to be performed at a time. The RSA algorithm [14] and the Diffie-Hellman key exchange scheme [4], and more recently the generalization of Paillier's probabilistic public-key scheme (with applications in private information retrieval) [3], require computing a sequence of modular multiplications of large integer operands, e.g. at least 1024 bits in length. Hence, efficient implementation of modular multiplication is crucial. For instance, in the RSA algorithm, an exponentiation is computed by a chain of modular multiplication and squaring operations [14]. Modular multiplication/squaring is normally achieved by an integer multiplication/squaring followed by a modular reduction by a predefined modulus.

In algorithms such as RSA, where the predefined modulus is a random number, the required modular reduction of the result of an integer multiplication is more costly than the multiplication itself. The Montgomery residue representation and the resulting Montgomery multiplication algorithm have proven useful in reducing this complexity [11, 9]. In this representation, modular reductions are partially avoided and thus the overall computation is simplified.

In Montgomery multiplication, firstly the operands are converted to their respective Montgomery residue representations, then the desired sequence of operations are performed using Montgomery multiplication, and finally the result is converted back to the normal integer representation. The Montgomery multiplication algorithm, given with Algorithm 1, computes $A{\cdot}B{\cdot}2^{-m}$ for the input operands A and B which are the Montgomery residue representations of the two integers X and Y such that $A = X \cdot 2^m$ and $B = Y \cdot 2^m$. Note that Algorithm 1 keeps the residue representation

intact, i.e., $A \cdot B \cdot 2^{-m} \equiv (X \cdot Y) \cdot 2^m \pmod{n}$ which allows for further computations avoiding extra operations.

Algorithm 1 explains the general Montgomery multiplication algorithm. A detailed analysis of different Montgomery multiplication algorithms can be found in [1]. Among these Montgomery multiplication algorithms, the Coarsely Integrated Operand Scanning (CIOS) Method is considered the fastest one in most processor platforms and described in Section 2.2.

Algorithm 1 Montgomery multiplication

Input: $A, B \in Z_n$ where n is an odd integer, $n' = -n^{-1} \bmod 2^m$ where $m = \lceil \log_2 n \rceil$.
Output: $A \cdot B \cdot 2^{-m} \bmod n$.

1: $t \leftarrow A \cdot B$
2: $t \leftarrow (t + (t \cdot n' \bmod 2^m) \cdot n)/2^m$
3: **if** $t \geq n$ **then**
4: Return $t - n$
5: **else**
6: Return t
7: **end if**

2.2 Coarsely Integrated Operand Scanning (CIOS) Method for Montgomery Multiplication

Among all the Montgomery multiplication algorithms listed in [1], the CIOS method, presented below with Algorithm 2, requires the least storage and has the best timing performance, and therefore it is the most preferred Montgomery multiplication algorithm.

Algorithm 2 CIOS method for Montgomery multiplication

Input: $A, B \in Z_n$ where n is an odd integer, $n' = -n^{-1} \bmod 2^{s \cdot w}$ where w is the processor word size and $s = \lceil \lceil \log_2 n \rceil / w \rceil$.
Output: $A \cdot B \cdot 2^{-s \cdot w} \bmod n$.

1: **for** $i = 0 \rightarrow s - 1$ **do**
2: $C \leftarrow 0$
3: **for** $j = 0 \rightarrow s - 1$ **do**
4: $(C, S) \leftarrow t_j + a_j \cdot b_i + C$
5: $t_j \leftarrow S$
6: **end for**
7: $t_s \leftarrow S$
8: $t_{s+1} \leftarrow C$
9: $C \leftarrow 0$
10: $m \leftarrow t_0 \cdot n'_0 \bmod 2^w$
11: $(C, S) \leftarrow t_0 + m \cdot n_0$
12: **for** $j = 1 \rightarrow s - 1$ **do**
13: $(C, S) \leftarrow t_j + m \cdot n_j + C$
14: $t_{j-1} \leftarrow S$
15: **end for**
16: $(C, S) \leftarrow t_s + C$
17: $t_{s-1} \leftarrow S$
18: $t_s \leftarrow t_{s+1} + C$
19: **end for**
20: **if** $[t_s \ t_{s-1} \ t_{s-1} \cdots t_0]_{2^w} \geq n$ **then**
21: Return $[t_s \ t_{s-1} \ t_{s-1} \cdots t_0]_{2^w} - n$
22: **else**
23: Return $[t_{s-1} \ t_{s-1} \cdots t_0]_{2^w}$
24: **end if**

All the Montgomery multiplication algorithms listed in [1], including the CIOS method, are word-based, inherently serial algorithms and do not allow parallelization in software realizations on general-purpose processors. In this work, we investigate the parallelization of Montgomery multiplication on general-purpose multi-core processors.

3 General-Purpose Multi-Core Architectures and Parallel Programming

The last decade experienced an important paradigm shift in processor design towards multi-core architectures. Hitting the so-called power wall and exhausting means for instruction level parallelism (ILP) as a result of decades long research and development [13] intensified the focus on the exploitation of thread-level parallelism through multi-core architectures which are made possible by the famous Moore's law. Nowadays, not only

desktop computers or workstations but also almost all notebook computers and even some smart phones are shipped with multi-core processors. In the foreseeable future, this trend will continue with ever increasing momentum and we will see *many-core* processors that feature tens of cores of identical general-purpose and/or different specialized architectures.

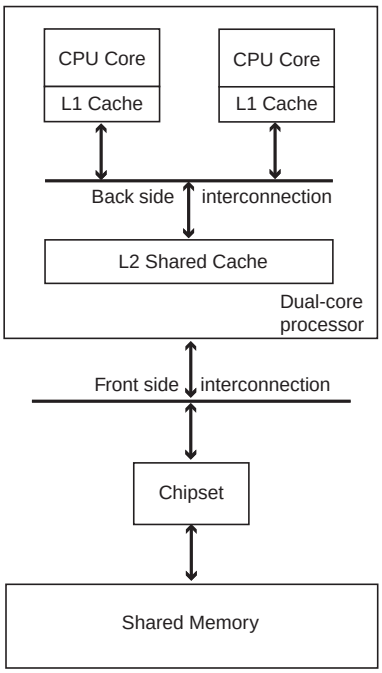

Fig. 1. Architecture of a dual-core general-purpose processor.

In Figure 1, a simple dual-core processor with two identical general-purpose processor cores is illustrated. Each core features a local cache memory which is referred as level one (L1) cache since it stands at the top of the memory hierarchy. The second level cache (L2 cache) and the main memory are shared by the processor cores. Many general-purpose multi-core architectures have similar organizations independent of the number of cores and cache levels. Therefore, the multi-core architectures are considered to be in the category of *shared memory* multiprocessor systems, where the cores are *synchronized* through the shared memory and do not interact directly otherwise. Cores operate on the local (private) data in their L1 cache memories independently when operations assigned to the

cores are independent. However, L1 caches are synchronized through *consistency protocols* [13], which are implemented in hardware and therefore transparent to application developer, when cores require each other's data or need to process the same shared data. An efficient parallel program tries to minimize the number of synchronization points allowing cores to work independently for durations as long as possible.

OpenMP (Open Multi-Processing)[4] provides the necessary application programming interface (API) for parallel programming in shared memory multi-processor systems. It features many library routines, compiler directives, and environment variables to support multi-threaded application development, whereby threads can be scheduled on individual cores by the developer. OpenMP API can be profitably utilized to accelerate cryptographic operation on multi-core processors. In this work, we utilize OpenMP for the parallel implementation of our proposed Montgomery multiplication algorithm.

4 Montgomery Multiplication Utilizing Multi Cores

All algorithms commonly proposed for Montgomery multiplication are word based algorithms, which perform the required partial product computations and modular reductions interleaved together and on a word by word basis, yielding the serial nature of these algorithms. In order to parallelize Montgomery multiplication, for two and three core architectures, bipartite [7, 8] and tripartite [16] Montgomery multiplication algorithms, respectively, were proposed. In [15, 2, 5], specialized multi-core hardware architectures are proposed for parallel implementation of the Montgomery multiplication algorithm. However these algorithms are intended for hardware based implementations and not targeted for general purpose microprocessors. In [2], the authors give a theoretical analysis of possible parallelizations of the SOS version of Montgomery multiplication given in [1], and implementation results on prototype multi-core systems using softcore processors on FPGA devices. The proposed design in [2] utilizes fast communication between the utilized softcores and local memories both of which are specifically tailored to the proposed parallel implementation of the Montgomery multiplication algorithm. Therefore, their approach represents a hybrid architecture that takes advantage of both software and hardware. In this section, we propose a novel parallel Montgomery multiplication algorithm which is specifically designed for software realizations

[4] OpenMP Tutorial at Supercomputing 2008, http://openmp.org/wp/2008/10/openmp-tutorial-at-supercomputing-2008/ (Last accessed on 26 February 2012.)

and thus is suitable for general-purpose microprocessors. For our parallel Montgomery multiplication algorithm, we exploit the inherent parallelism in integer multiplication.

4.1 Inherent Parallelism in Integer Multiplication

Remember the integer multiplication operation shown in Figure 2. As shown in Figure 2, integer multiplication has an inherent parallelism which could be exploited by running the multiple cores available on a processor in parallel. Note in Figure 2 that the partial products required for integer multiplication can be computed in parallel and then accumulated to give us the actual product.

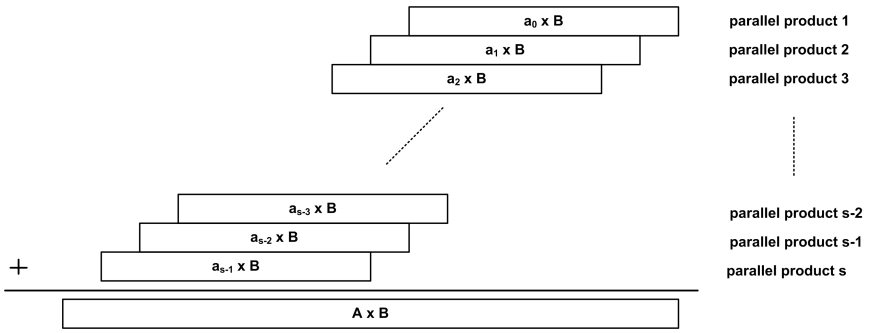

Fig. 2. Inherent parallelism in the integer multiplication $A \times B$. Note that A can be represented as $[a_{s-1}\ a_{s-2}\ \cdots\ a_0]$ in base $2^{\left\lceil \frac{\log_2 A}{s} \right\rceil}$.

We give the following algorithm for parallel integer multiplication on general-purpose multi-core processors.

4.2 Parallel Montgomery Multiplication

We adapt the inherent parallelism of Algorithm 3 to the original Montgomery multiplication algorithm given with Algorithm 1 for application on multi-core processors. The resulting parallel Montgomery multiplication algorithm is given with Algorithm 4 below.

On a multi-core processor, one can also parallelize the additions given on lines 4 to 6 (required for the accumulation of the partial products) of Algorithm 3. When the number of cores available is a power of 2, this

Algorithm 3 Parallel Integer Multiplication

Input: Integers $A = [a_{s-1}\ a_{s-2}\ \cdots\ a_0]_d$ and B of size $m = d \cdot s$ bits where s is the number of cores available.
Output: **ParallelMultiply**$(A, B) = A \cdot B$.

1: **for** $i = 0$ to $s - 1$ **do**
2: $t_i \leftarrow a_i \cdot B \cdot 2^{i \cdot d}$ {performed at core $i + 1$ in a multi-core implementation}
3: **end for**
4: **for** $i = 1$ to $s - 1$ **do**
5: $t_0 \leftarrow t_0 + t_i$
6: **end for**
7: Return (t_0)

Algorithm 4 Parallel Montgomery multiplication

Input: $A, B \in Z_n$ where n is an odd integer and $n' = -n^{-1} \bmod 2^m$ where $m = \lceil \log_2 n \rceil$.
Output: $A \cdot B \cdot 2^{-m} \bmod n$.

1: $t \leftarrow$ **ParallelMultiply**(A, B) {Algorithm 3}
2: $u \leftarrow$ **ParallelMultiply**$(t, n') \bmod 2^m$ {Algorithm 3}
3: $u \leftarrow$ **ParallelMultiply**(u, n) {Algorithm 3}
4: $u \leftarrow (u + t)/2^m$
5: **if** $u \geq n$ **then**
6: Return $(u - n)$
7: **else**
8: Return (u)
9: **end if**

partial product accumulation can be achieved in a binary tree fashion, as shown below, with at most $\lceil \log_2 s \rceil$ steps where s is the number of cores available.

```
for i = 1 to log2 s
        for j = 0 to s/2^i − 1
                t_j ← t_j + t_{j+s/2^i}
        end for
end for
```

In the above setting, all the cores are exploited as evenly as possible with the maximal utilization which would result in the minimal latency. However, this optimal chain of additions would not always be possible. In the rest of this section, we provide some addition chains for efficient implementations of Algorithm 4 on processors with 2, 4 and 6 cores as examples.

Partial Product Accumulation on a 2-Core Processor:

For performing the integer multiplication $A \times B$ on 2 cores, the operand A is divided into two equal parts as $[a_1\ a_0]$ in base $2^{\left\lceil \frac{\log_2 A}{2} \right\rceil}$ and the partial products $a_0 \times B$ and $a_1 \times B$ are computed simultaneously on separate cores. Finally, these partial products are accumulated as given below.

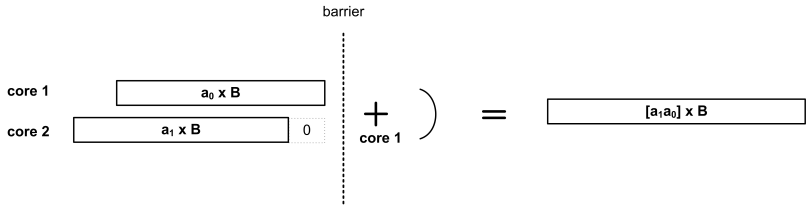

Fig. 3. Computation and accumulation of the partial products for the integer multiplication $A \times B$ on 2 cores. Note that A is divided into two equal parts and denoted as $[a_1\ a_0]$ in base $2^{\left\lceil \frac{\log_2 A}{2} \right\rceil}$.

Partial Product Accumulation on a 4-Core Processor:

On a 4-core processor, for performing the integer multiplication $A \times B$, the operand A is divided into four equal parts as $[a_3\ a_2\ a_1\ a_0]$ in base

$2^{\left\lceil \frac{\log_2 A}{4} \right\rceil}$ and the partial products $a_0 \times B$, $a_1 \times B$, $a_2 \times B$ and $a_3 \times B$ are computed simultaneously on separate cores. Finally, these partial products are accumulated with the optimal addition chain given below.

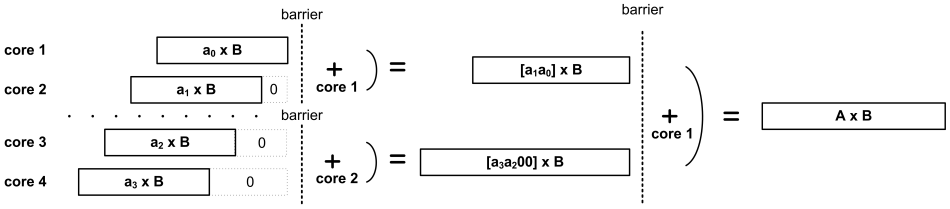

Fig. 4. Computation and accumulation of the partial products for the integer multiplication $A \times B$ with 4 cores. A is divided into four equal parts and denoted as $[a_3\ a_2\ a_1\ a_0]$ in base $2^{\left\lceil \frac{\log_2 A}{4} \right\rceil}$.

Partial Product Accumulation on a 6-Core Processor:

On 6-core processor, for performing the integer multiplication $A \times B$, the operand A is divided into six equal parts as $[a_5\ a_4\ a_3\ a_2\ a_1\ a_0]$ in base $2^{\left\lceil \frac{\log_2 A}{6} \right\rceil}$ and the partial products $a_0 \times B$, $a_1 \times B$, $a_2 \times B$, $a_3 \times B$, $a_4 \times B$ and $a_5 \times B$ are computed simultaneously on separate cores. Finally, these partial products are accumulated with the addition chain given below.

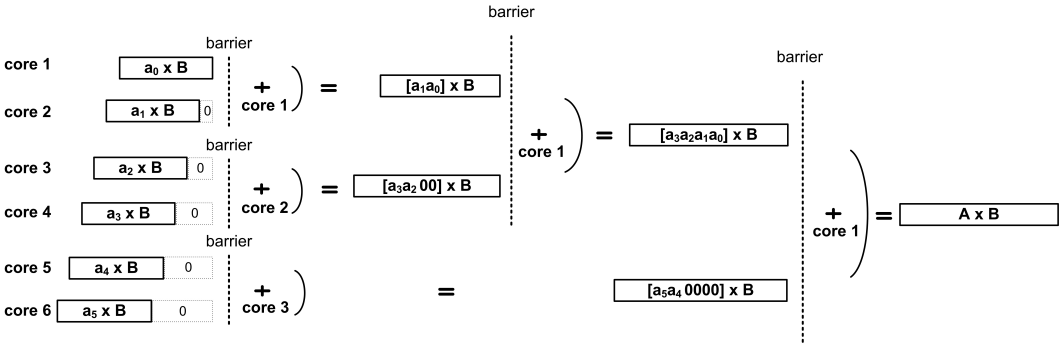

Fig. 5. Partial product computation and accumulation of the partial products for the integer multiplication $A \times B$ with 6 cores. A is divided into six equal parts and denoted as $[a_5\ a_4\ a_3\ a_2\ a_1\ a_0]$ in base $2^{\left\lceil \frac{\log_2 A}{6} \right\rceil}$.

5 Timing Performance

We implemented our algorithm for the operand sizes of 1024, 2048, 4096, 8192, 16384 and 32768 bits on general-purpose multi-core processors using OpenMP and obtained the timings. We used 2, 4 and 6 core general-purpose processors for our implementations and made use of the efficient addition chains given with Figures 3, 4 and 5, respectively. The timing graphs for the implementation of our proposed Montgomery multiplication algorithm, as well as the CIOS method, on 2, 4 and 6 core general-purpose processors can be seen in Figures 6, 7 and 8, respectively. Detailed timings and achieved speedups (compared to the CIOS method) can be found in Tables 1, 2 and 3 (in Appendix).

We observe in Figures 3, 4 and 5 that our algorithm performs significantly better than the CIOS method, and furthermore efficiently utilizes multiple cores for improved performance, for growing operand sizes. As seen in Tables 1, 2 and 3, it achieves up to 81%, 3.37 times and 4.87 times speedups for the used general-purpose microprocessors with 2, 4 and 6 cores, respectively. For the operand sizes of 2048 bit and smaller, the multi-core performance of our algorithm is worse than its single-core performance due to the overhead from using the OpenMP library.

On the 2 and 4 core processors, the single-core performance of our algorithm is better than the CIOS method for operand sizes of 2048 bit and larger (see Tables 1 and 2 in Appendix). Whereas, on the 6 core processor, the single-core performance of our algorithm performs better starting with the larger operand size of 4096 bit (see Table 3 in Appendix).

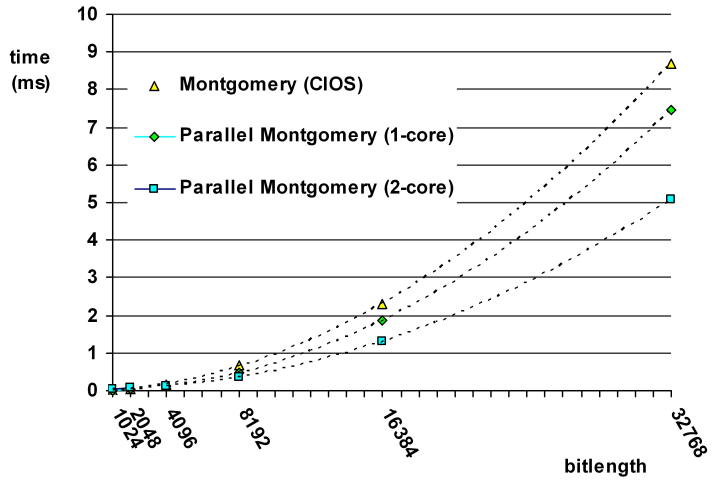

Fig. 6. Timings for the CIOS and Parallel Montgomery multiplication algorithms on the 2-core Intel Dual-Core Pentium E6500 processor running at 2.93 GHz and with 2.96 GB of RAM.

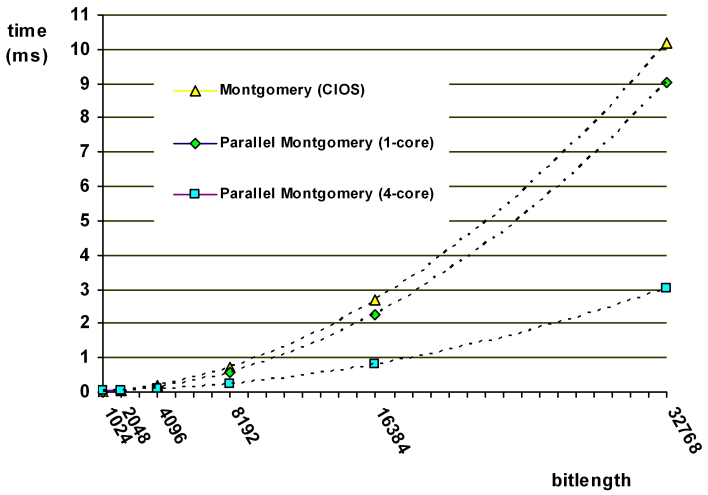

Fig. 7. Timings for the CIOS and Parallel Montgomery multiplication algorithms on the 4-core Intel Core 2 Quad processor Q8300 running at 2.5 GHz and with 4 GB of RAM.

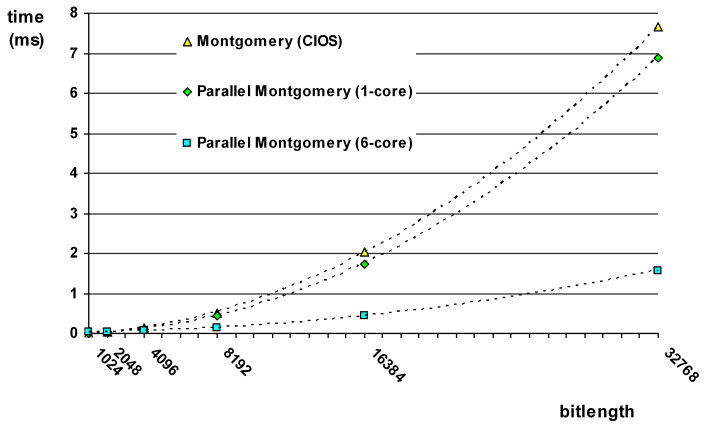

Fig. 8. Timings for the CIOS and Parallel Montgomery multiplication algorithms on the 6-core Intel Xeon W3670 processor running at 3.2 GHz and with 8 GB RAM.

This discrepancy is most possibly due to the fact that the addition chains used for the partial product accumulations on 2 and 4 core processors (given in Figures 3 and 4, respectively) are optimal whereas the one used for the 6-core processor (as given in Figure 5) is not optimal. On the used 2, 4 and 6 core general-purpose processors, the single-core performance of our algorithm is up to 39%, 26% and 17% better, respectively, compared to the CIOS method.

6 Conclusion

In this work, we presented for the first time an efficient parallel Montgomery multiplication algorithm for software implementations on general-purpose multi-core processors. Our algorithm exhibits good timing performance in both single-core and multi-core implementations. We identify the utilization of our algorithm for efficient implementation of classical cryptographic schemes such as RSA and Diffie-Hellman, as well as more advanced schemes such the Damgård-Jurik's algorithm, as future work.

References

1. Ç. K. Koç, T. Acar, and B. Kaliski. Analyzing and comparing montgomery multiplication algorithms. *IEEE Micro*, pages 26–33, 1996.

2. Zhimin Chen and Patrick Schaumont. A parallel implementation of montgomery multiplication on multicore systems: Algorithm, analysis, and prototype. *IEEE Trans. Comput.*, 60:1692–1703, December 2011.
3. Ivan Damgård and Mats Jurik. A generalisation, a simplification and some applications of paillier's probabilistic public-key system. In *Proceedings of the 4th International Workshop on Practice and Theory in Public Key Cryptography: Public Key Cryptography*, PKC '01, pages 119–136, London, UK, 2001. Springer-Verlag.
4. W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22:644–654, 1976.
5. Junfeng Fan, Kazuo Sakiyama, and Ingrid Verbauwhede. Montgomery modular multiplication algorithm on multi-core systems. *2007 IEEE Workshop on Signal Processing Systems*, 10:261–266, 2007.
6. Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. i-hop homomorphic encryption and rerandomizable yao circuits. In Tal Rabin, editor, *CRYPTO*, volume 6223 of *Lecture Notes in Computer Science*, pages 155–172. Springer, 2010.
7. Marcelo E. Kaihara and Naofumi Takagi. Bipartite modular multiplication. In *Proceedings of Cryptographic Hardware and Embedded Systems - CHES 2005, number 3659 in Lecture notes in Computer Science*, pages 201–210. Springer-Verlag, 2005.
8. Marcelo E. Kaihara and Naofumi Takagi. Bipartite modular multiplication method. *IEEE Transactions on Computers*, 57(2):157–164, 2008.
9. Ç. K Koç and T. Acar. Montgomery Multplication in $GF(2^k)$. *Design, Codes, and Cryptography*, 14(1):57–69, 1998.
10. Helger Lipmaa. First CPIR protocol with data-dependent computation. In *Proceedings of the 12th international conference on Information security and cryptology*, ICISC'09, pages 193–210, Berlin, Heidelberg, 2010. Springer-Verlag.
11. P. L. Montgomery. Modular Multiplication without Trial Division. *Mathematics of Computation*, 44(170):519–521, April 1985.
12. Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *IN ADVANCES IN CRYPTOLOGY - EUROCRYPT 1999*, pages 223–238. Springer-Verlag, 1999.
13. David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware / Software Interface*. Morgan Kaufmann, Elsevier, 4^{th} edition, 2012. (Revised printing).
14. R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.
15. Kazuo Sakiyama, Lejla Batina, Bart Preneel, and Ingrid Verbauwhede. Multicore Curve-Based Cryptoprocessor with Reconfigurable Modular Arithmetic Logic Units over $GF(2^n)$. *IEEE Transactions on Computers*, 56:1269–1282, 2007.
16. Kazuo Sakiyama, Miroslav Knezevic, Junfeng Fan, Bart Preneel, and Ingrid Verbauwhede. Tripartite modular multiplication. *Integration*, 44(4):259–269, 2011.

A Timings

		Operand Size (in bits)					
		1024	2048	4096	8192	16384	32768
Algorithm			**time (ms)**				
Montgomery-CIOS (Single Core)		0.0091	0.0341	0.1488	0.6564	2.3043	8.7010
Parallel Montgomery on Single Core		0.0121	0.0321	0.1222	0.4708	1.8675	7.4530
	Speedup:	**−25%**	**6%**	**22%**	**39%**	**23%**	**17%**
Parallel Montgomery on 2 Cores (with OpenMP support)		0.0372	0.0635	0.1227	0.3620	1.2953	5.0640
	Speedup:	**−75%**	**−46%**	**21%**	**81%**	**78%**	**72%**

Table 1. Timings (in ms) for the CIOS and Parallel Montgomery multiplication algorithms on the 2-core Intel Dual-Core Pentium E6500 processor running at 2.93 GHz and with 2.96 GB of RAM.

		Operand Size (in bits)					
		1024	2048	4096	8192	16384	32768
Algorithm			**time (ms)**				
Montgomery-CIOS (Single Core)		0.0103	0.0402	0.1692	0.7321	2.6929	10.1625
Parallel Montgomery on Single Core		0.0111	0.0382	0.1499	0.5835	2.2798	9.0456
	Speedup:	**−7%**	**5%**	**13%**	**26%**	**18%**	**12%**
Parallel Montgomery on 4 Cores (with OpenMP support)		0.0448	0.0554	0.0866	0.2394	0.8044	3.0140
	Speedup:	**−77%**	**−27%**	**95%**	**×3.06**	**×3.35**	**×3.37**

Table 2. Timings (in ms) for the CIOS and Parallel Montgomery multiplication algorithms on the 4-core Intel Core 2 Quad Processor Q8300 running at 2.5 GHz and with 4 GB of RAM.

		Operand Size (in bits)					
		1024	2048	4096	8192	16384	32768
Algorithm			**time (ms)**				
Montgomery-CIOS (Single Core)		0.0077	0.0298	0.1267	0.5168	2.0259	7.6476
Parallel Montgomery on Single Core		0.0102	0.0314	0.1152	0.4432	1.7449	6.8921
	Speedup:	**−24%**	**−5%**	**10%**	**17%**	**16%**	**11%**
Parallel Montgomery on 6 Cores (with OpenMP support)		0.0432	0.0381	0.0682	0.1331	0.4390	1.5712
	Speedup:	**−82%**	**−22%**	**86%**	**×3.88**	**×4.62**	**×4.87**

Table 3. Timings for CIOS and Parallel Montgomery multiplication algorithms on a 6-core Intel Xeon W3670 processor running at 3.2 GHz and with 8 GB RAM.