A Parallel Implementation of Montgomery Multiplication on Multi-core Systems: Algorithm, Analysis, and Prototype

Zhimin Chen, Student Member, IEEE, and Patrick Schaumont, Senior Member, IEEE

Abstract—The Montgomery Multiplication is one of the cornerstones of public-key cryptography, with important applications in the RSA algorithm, in Elliptic-Curve Cryptography, and in the Digital Signature Standard. The efficient implementation of this long-wordlength modular multiplication is crucial for the performance of public-key cryptography. Along with the strong momentum of shifting from single-core to multi-core systems, we present a parallel software implementation of the Montgomery multiplication for multi-core systems. Our comprehensive analysis shows that the proposed scheme, pSHS, partitions the task in a balanced way so that each core has the same amount of job to do. In addition, we also comprehensively analyze the impact of inter-core communication overhead on the performance of pSHS. The analysis reveals that pSHS is high-performance, scalable over different number of cores, and stable when the communication latency changes. The analysis also tells us how to set different parameters to achieve the optimal performance. We implemented pSHS on a prototype multi-core architecture configured in a Field Programmable Gate Array (FPGA). Compared with the sequential implementation, pSHS accelerates 2048-bit Montgomery multiplication by 1.97, 3.68, and 6.13 times on respectively 2-core, 4-core, and 8-core architectures with communication latency equal to 100 clock cycles.

Index Terms—Montgomery multiplication, public-key cryptography, parallel programming, tiled processor.

1 INTRODUCTION

MULTI-CORE architectures are replacing single-core architectures at a rapid pace. Indeed, multi-core architectures offer better performance and energy-efficiency for the same silicon footprint [1]. As a result, multi-core architectures can now be found in a wide range of system architectures, including servers, desktops, embedded and portable architectures [2, 3, 4, 5].

Parallel software development is a major challenge in the transition of single-core to multi-core architectures. Programming environments such as OpenMP or MPI provide a parallel programming model to abstract the underlying parallel architecture. A programmer then has to express a sequential algorithm as parallel, communicating tasks. Under ideal circumstances, a parallel version of a sequential program will run \( N \) times faster on \( N \) processors, yielding a speedup of \( N \). In practice, the actual speedup is less, because of unbalanced partitioning of the algorithm, and because of inter-processor communication dependencies. Parallel-software programmers therefore try to balance the computational load as much as possible, and to minimize the effect of inter-processor communication.

In this paper, we investigate how to parallelize the Montgomery Multiplication (MM) [6]. This is an algorithm for modular multiplication that is extensively used in public-key cryptography. In practice, the wordlength of such MM is many times larger than the wordlength of the underlying processor. This leads to multi-precision arithmetic. For example, a direct implementation of a 2048-bit multiplication on a 32-bit processor will require 4096 32-bit multiplications just to obtain all the partial products. This is because a multiplication has quadratic complexity. For modular multiplications, the complexity is even higher because the reduction (modular-part) needs to be implemented as well. Koç summarized several sequential solutions for multi-precision MM [7].

Several authors have explored straightforward parallelization of the MM by executing multiple independent MM on multiple cores in parallel [8, 9, 10, 11, 12]. This scenario is a fixed-latency scheme: it optimizes the throughput (MMs completed per second), but it does not enhance the latency of a single MM (time to finish one MM). Public-key operations such as RSA [13], DSA [14], and ECC [15] contain thousands of MMs. However, these MMs contain strong data dependencies, and the result of one MM may be needed as an operand for the next MM. A high-throughput, fixed latency scheme for MM therefore cannot accelerate a single instance of RSA, DSA, or ECC. In an embedded-computing context, where a single user is waiting for the result of a single public-key operation, a low-latency scheme is needed instead. In the development of a low-latency MM, we are looking for the following properties.

- Balanced task partitioning. Partitioning divides one MM into several different parts and assigns them to different cores for computation. A balanced task partitioning means that each core has the same task load. Better balancing will improve the potential...
speedup that the parallel program can achieve.
- **High tolerance to the communication delay.** Running one MM with multiple cores inevitably leads to inter-core communication. Inter-core communication is usually much slower than within-core communication. Therefore, inter-core communication is a likely bottleneck in parallel solutions. A high tolerance to the communication delay means that the performance of the parallel solution is not severely influenced by inter-core communications. This feature makes it easier to port the parallel MM to different multi-core architectures and stabilizes the performance as the communication delay varies.
- **High scalability.** A good parallel design should be able to handle variations of the algorithm as well as of the target architecture. At the algorithm level, the design parameters of the public-key algorithms may vary (RSA-1024 and RSA-2048, for example). Also, future multi-core systems can be expected to have a larger number of cores than the systems of today, as a result of increased integration. Therefore, we are looking for a scalable, parallel implementation of the MM.

Several implementations of a parallel MM have been presented before, though we believe that none of these meets all the requirements we enumerated above. Bajard et al. proposed to formulate the MM using a residue number system (RNS) [16]. An RNS allows to break the long-wordlength operation of the MM into a set of smaller numbers, which then leads to a set of parallel operations. However, the resulting design is not fully balanced over the complete set of processors. We will show that we can achieve a multi-precision implementation of MM that is, from a computational perspective, perfectly balanced over all processors.

Another parallelization of MM was proposed by Kairhe et al. with the Bipartite Montgomery formulation [17, 18]. A bipartite design breaks a MM in two parts, so that two processors can compute a MM in parallel. This solution cannot be easily scaled to multiple cores, since each part still runs on a single processor. Enhancements for further parallelization, such as the tripartite MM by Sakiyama et al., show that this parallelization is quite complex and the scalability is still limited [19].

Sakiyama et al. and Fan et al. describe parallel, scalable implementations of the MM using custom-designed processors [20, 21, 22]. Such implementations are not portable, and cannot be expressed on top of OpenMP or MPI because the underlying programming language does not support the specialized operators used in their designs. The same problem can also be found with an implementation of arithmetic modulo minimal redundancy Cyclotomic Primes for ECC by Baldwin et al. [23]. In contrast, we seek a portable, software-only formulation of the MM.

**Our Contribution:** We propose a parallelization of the MM called parallel Separated Hybrid Scanning (pSHS). Instead of turning to the algorithm conversions like RNS and bipartite MMs, pSHS directly partitions a regular MM in such a way that each processing core has the same computational load. This partitioning method can scale over different numbers of cores. In addition, we present a detailed analysis of the inter-core communication overhead of pSHS, and we conclude that this algorithm has a high tolerance to inter-core communication delays. We will show that pSHS has the 3 properties listed above: balanced task partitioning, high tolerance to communication delay, and high scalability.

We have built multi-core prototypes with 2, 4 and 8 soft-cores on an FPGA. Through our experiments, we see that pSHS offers a considerable speedup over the sequential solution and it is easy to scale over different numbers of cores. For example, a parallel 2048-bit MM implemented on 32-bit embedded cores has speedups of 1.97, 3.68, and 6.13 based on 2-, 4-, and 8-core architectures respectively when the inter-core communication latency is as high as 100 clock cycles. The efficiencies per core are as high as 0.99, 0.94, and 0.82 respectively. We also show that integrating pSHS to RSA does not lose any of pSHS’s advantages. The parallelized 2048-bit RSA encryption gains almost the same speedups as pSHS. The multi-core prototypes on the FPGA represent a category of multi-core architectures, where different cores run independently with the communication handled by a message-passing network. We expect the results obtained from our experiments to be portable to the similar multi-core architectures, for example the tiled processors [24, 25, 26, 27, 28].

This paper is an extension of the previous work [29]. Compared to the earlier work, this paper provides additional motivation on the task partitioning used for pSHS, it provides a formal analysis of pSHS, and it demonstrates the integration of pSHS into RSA.

## 2 Sequential Montgomery Multiplication and Sequential Schemes

Intuitively, modular multiplication requires division operations with arbitrary dividers which are hard to implement. MM can replace the modular-$n$ operation with a division of the power of 2 ($2^m$). Thus, the division can be achieved by simple shifting, which is easy to implement in processors. This makes MM one of the most prevalent ways to implement modular multiplications and exponentiations. One of its sequential implementation schemes is shown in Algorithm 1 [7]. The most time-consuming part of Algorithm 1 is from line 2 to line 14. Our research will focus on this part and ignore the final subtraction in line 15 and 16, since it does not affect the overall performance too much. In addition, Walter presented Montgomery Exponentiation without final subtractions in MM by changing the bound of MM’s operands [30]. Such case, the influence of the final subtraction can be completely ignored. Considering the final subtraction has a very small influence to the overall performance, although the following discussion depends...
Algorithm 1 Montgomery multiplication (SOS scheme) [7]

Require: An $s$-word modulus $N = (n_{s-1}, n_{s-2}, ..., n_1, n_0)$, two operands $A = (a_{s-1}, a_{s-2}, ..., a_1, a_0)$, and $B = (b_{s-1}, b_{s-2}, ..., b_1, b_0)$ with $A, B < N$, $A, B, N > 0$, $N$ being odd, and the constant $n' = -n_0^2 \mod 2^w$. $w$ is the word length of a system (usually 8, 16, 32, or 64). $T = (t_{2s-1}, t_{2s-2}, ..., s, t_0)$ is a temporary array. * means integer multiplication.

Ensure: $R = (r_{s}, r_{s-1}, ..., r_1, r_0) = A \times B \times 2^{w} \mod N$. $n = w \times s$.

1: $T = 0$
2: for $i = 0$ to $s - 1$ do
3: $C = 0$
4: for $j = 0$ to $s - 1$ do
5: $(C, S) = t[i] + j + a[j] \times b[i] + C$
6: $t[i + j] = S$
7: $t[i] + s = C$
8: for $i = 0$ to $s - 1$ do
9: $C = 0$
10: $m = t[i] \times n' \mod 2^w$
11: for $j = 0$ to $s - 1$ do
12: $(C, S) = (t[i] + j + m \times n[j] + C$
13: $t[i] + j = S$
14: ADD ($t[i] + s, C$) \{addition and propagate carry to higher part of $T$\}
15: $R = (t_s, t_{s-1}, ..., t_1, t_0)$
16: if $R > N$ then
17: $R = R - N$

Fig. 1. (a) Analysis model: boxes; (b) Data flow of Montgomery multiplication

3 Parallel Schemes

3.1 Multi-core Model

To make our programming scheme portable, we base our parallel scheme on a general message-passing multicore model as follows. In the multicore architecture, processing cores work independently with their own local memory. An on-chip network connects the cores together. Different cores communicate with each other by message passing through the network. When CORE0 needs to transfer data to CORE1, it packages the data (one or several words) as a message, sends it to the network and moves on. The message will go through the network and be stored in CORE1’s local memory until CORE1 reads it. If the message arrives later than CORE1’s read operation, CORE1 will wait or be stalled until the message arrives.

The communication latency is critical for the overall performance of a parallel system. In the analysis, we consider the worst case and use the longest communication latency for every message transfer and define it as transfer time unit (TTU). A parallel programming scheme that obtains satisfying analysis results from the worst case will yield good performance on many real multicore systems as well.
3.2 Comparing row-based and column-based partitioning

In order to obtain an efficient parallel program, we have to achieve two goals: balanced task partitioning and reduction of communication and synchronization cost. Montgomery Multiplication consists of both the regular operations, including the boxes, and the irregular operations, including generation of 'm' and the final subtraction. While it is easy to partition the process of the boxes, partitioning the irregular operations is not that easy. Fortunately, the irregular operations occur much less frequently and cost much less time than the regular operations. Therefore, we approximate a balanced task division to be assigning the same amount of boxes to different processors. Reduction of communication means less data dependency among different processors. Synchronization cost is the stalled time of a processor, which can be caused by the data dependency and the communication delay.

3.2.1 Row-based partitioning

One obvious method for task division is to assign different rows of boxes from the top to the bottom to different processors in a circular way. This method is illustrated in Figure 2(a), in which eight interleaved rows of boxes are handled by 4 cores (P0 to P3). To make the processors more synchronized, an ‘integrated’ method is used. This kind of task division follows the idea of operand scanning: each processor calculates boxes in each row from the right to the left.

Task partitioning. If the number of rows (2 * s) can be divided by the number of processors (p), then every processor is assigned with the same amount of boxes, which achieves a balanced task partitioning.

Communication. Based on the row-based partitioning, c and m stay at the same row while t goes from one row to another, which means two neighboring cores need to transfer t. Since t is the result of a box, it can only be transferred after the box’s calculation is finished. We draw arrows to represent transferring t. For each processor, there are s – 1 or s words of t and the final carry word for ‘send’ and ‘receive’. In total, the number of communications counts for approximately 2 * s^2 / p per processor.

3.2.2 Column-based partitioning

Another method to partition the task is according to columns. This kind of task partitioning comes from the ‘product scanning’ method. The approach is to assign columns of boxes, from the right to the left, to processors in a circular way, shown in Figure 2(b).

Task partitioning. The entire task can be divided into two parts: the first half from column 0 to column s-1 and the second from column s to column 2 * s – 1. We found that these two parts are complementary - shifting the second half to the right by s columns gives us a perfect rectangular. If the number of columns in the first half (s) can be divided by the number of processors (p), then every processor is assigned with the same amount of boxes, which leads to a balanced task partitioning.

Communication. In this method, m for a row of shaded boxes is generated in one processor P_i and then transferred to the nearest box in the same row that is assigned to the next processor P_{i+1}. After that, m is transferred in the same way until it reaches a box that is assigned to P_i again. Besides m, c also needs to be transferred. The first impression is that column-based division requires more data communication. However, improvements can be done to accumulate all the carries in one column to form a final one (two words) and then transfer it to the next column. We represent the transfer of m and c by arrows from a box to another in Figure 2(b). Every time when a processor handles a column, it needs to transfer the words of m generated by the previous p – 2 processors as well as the words of m generated by itself. Plus, 2 words of c are transferred.
Thus, the number of data communication operations is 
\((p - 1)s/p + 2 \times 2s/p = s(p + 3)/p\) per processor.

Both of the above two schemes only require communication between neighboring processors. By comparison, we find that the column-based scheme needs less communication when \(p < 2s - 3\), which usually holds true. Therefore, we base our proposed parallel programming scheme on column-based partitioning.

We also note that the Karatsuba optimization [31] is a well-known partitioning of long-wordlength multiplication. Although it leads to a sub-quadratic increase of the amount of multiplications, the Karatsuba algorithm also has a super-quadratic increase in the number of accumulations. In parallel software, where the cost of an addition and a multiplication is similar, Karatsuba optimization therefore does not lead to an obvious advantage. We do not take Karatsuba optimization into account in this paper.

### 3.3 parallel Separated Hybrid Scanning (pSHS)

We first use a small example to explain the overall idea of pSHS and then generalize it to a formal algorithm. In Figure 3, we show the example of a 6-word MM with 12 rows and 12 columns of boxes processed by 3 cores. We do the task partitioning based on columns. Two adjacent columns of boxes are grouped into a Task Block (TB). Task Blocks are assigned to cores in a circular fashion. From the right to the left, Task Blocks are indexed from TB[0] to TB[5]. Core P0 starts with TB[0] and then continues with TB[3]. Similarly, P1 first handles TB[1] and then TB[4]. Inside each TB, one core first processes all the white boxes and then the shaded boxes. We put a number in the upper right corner of each box to indicate its execution order. The operation inside one TB is similar to the Separated Operand Scanning (SOS) scheme in [7].

On the other hand, the overall task partitioning is similar to Product Scanning. Therefore, we refer to our scheme as parallel Separated Hybrid Scanning (pSHS).

To integrate timing into the analysis model, we define the top of a box as the beginning of the calculation and the bottom as the end. All data communications between different TBs are shown as arrows in Figure 3. Each arrow represents one transfer operation. \(m_i\) has only one word and is sent out or required right before the rightmost shaded box of a row in a TB. As a result, one arrow for \(m_i\) starts from the top of a shaded box \((i,j)\) and points to the top of the shaded box \((i,j+q)\). The carry \(c_i\) has two words. We use two transfer operations. Since \(c_i\) is generated after the last shaded box \((k,l)\) of each TB and required by the next TB after shaded box \((k,l+q)\), we draw two arrows for \(c_i\) from the bottom of shaded box \((k,l)\) to the bottom of the shaded box \((k,l+q)\) or the top of shaded box \((k+1,l+1)\).

The last boxes for P0, P1 and P2 share the same label: 23. This shows that the task loads for different cores are the same (24 boxes). pSHS chooses column-based rather than row-based partitioning because this results in fewer messages between different TBs. This is because carries \((c)\) of different rows in a TB can be accumulated and sent to the next TB at the end of the current TB.

To generalize pSHS scheme, we define the meanings of symbols that we will use in Table 1. In an \(s\)-word

| \(w\) | Word length of a processing core (usually 8, 16, 32, or 64); |
| \(s\) | Number of words in the multiplication operands or the modulus; for a 512-bit Montgomery multiplication on a 32-bit processor, \(s = 16\); \(s\) can be divided by \(p \times q\); |
| \(p\) | Number of cores used for pSHS in the multicore system; |
| \(P\) | Order number of a core (from 0 to \(p - 1\)); |
| \(q\) | Number of columns of boxes in one TB. |

#### Algorithm 2 parallel Separated Hybrid Scanning (pSHS)

**Require:** An \(s\)-word modulus \(N = (n_{s-1}, n_{s-2}, ..., n_1, n_0)\), two operands \(A = (a_{s-1}, a_{s-2}, ..., a_1, a_0)\), and \(B = (b_{s-1}, b_{s-2}, ..., b_1, b_0)\) with \(A, B < N\), \(A, B, N > 0\), \(N\) being odd, and the constant \(n' = n_{s-1}^{-1} \pmod{2^w}\). \(w\) is the word length of a system (usually 8, 16, 32, or 64). \(s\) can be divided by \(p \times q\). \(N, A, B\) and \(B\) are stored locally for each core. \(RT = (rt_{q+1}, rt_{q}, ..., rt_1, rt_0)\) is a temporary array. \(*\) means integer multiplication.

**Ensure:** \(T = (t_{s-1}, t_{s-2}, ..., t_1, t_0) = A \times B \times 2^{-n} \pmod{N}\). \(n = w \times s\). \(T\) is stored locally for each core. The complete Montgomery Multiplication consists of \(p\) executions of the following program. On processor \(P\), execute:

1: for \(l = 0\) to \((2 \times s)/(p \times q) - 1\) do \{every iteration handles one TB\}
2: \(k = P_s * q + l * p * q;\)
3: \(p_{\text{pre}} = (P_l - 1) \pmod{p};\) \(p_{\text{next}} = (P_l + 1) \pmod{p};\)
4: initialize \(RT\) to 0;
5: for \(i = \max(0, k - s + 1)\) to \(\min(k + q, s) - 1\) do \{white box\}
6: \(ca = 0;\)
7: \(for j = \max(0, k - i)\) to \(\min(k - i + q, s) - 1\) do \{white box\}
8: \(\{ca, rt[j + k]\} = a[j] \times b[j] + rt[j + k] + ca;\)
9: \(h = \min(q, s - k + i);\)
10: \(rt[h] = ca + rt[h];\)
11: \(for i = \max(0, k - s + 1)\) to \(\min(k + q, s) - 1\) do \{shaded box\}
12: \(if i >= k then \{\text{12-18 generate and communicate}\}\)
13: \(m[i] = n' \times t[k - i];\) \(\text{send}(p_{\text{pre}} - p_{\text{next}}, m[i]);\)
14: else
15: \(if i >= k - p - (p - 1) \times q then \{\text{shaded box}\}\)
16: \(m[i] = \text{receive}(p_{\text{pre}} - p_{\text{next}});\)
17: \(if i >= k - p - (p - 2) \times q then \{\text{shaded box}\}\)
18: \(\text{send}(p_{\text{pre}} - p_{\text{next}}, m[i]);\)
19: \(ca = 0;\)
20: \(for j = \max(0, k - i)\) to \(\min(k - i + q, s) - 1\) do \{white box\}
21: \(\{ca, rt[j + k]\} = n[i] \times m[i] + rt[j + k] + ca;\)
22: \(h = \min(q, s - k + i);\)
23: \(rt[h] = ca + rt[h];\)
24: \(if i = \min(k-1, s-1) \text{ and } k \neq 0 \text{ then } \{24-7\ \text{communicate}\}\)
25: \(\text{receive}(p_{\text{pre}} - p_{\text{next}}, e[0]);\) \(\text{receive}(p_{\text{pre}} - p_{\text{next}}, e[1]);\)
26: \(\text{ADD}(rt[q + 1] \to 0, e[1] \to 0);\)
27: \(\text{send}(p_{\text{pre}} - p_{\text{next}}, rt[q]);\) \(\text{send}(p_{\text{pre}} - p_{\text{next}}, rt[q + 1]);\)
28: \(if k > s then \{\text{shaded box}\}\)
29: \(T[k - s + q - 1 \to k - s] = rt[q - 1 \to 0];\)
30: \(if P_0 = 0 \text{ then } \{\text{shaded box}\}\)
31: \(P_{\text{pre}} = \text{receive}(p_{\text{pre}} - p_{\text{next}});\)
32: \(\text{store}(rt[s/p - 1 \to 0] \to T \text{ stored locally}; \{\text{share results}\}\)
33: \(rt[s/p - 1 \to 0] = \text{receive}(p_{\text{pre}} - p_{\text{next}});\)
34: \(\text{store}(rt[s/p - 1 \to 0] \to T \text{ in the correlated location}; \{\text{share results}\}\)
35: \(if k = 0 \text{ to } p - 3 \text{ do } \{\text{shaded box}\}\)
36: \(\text{send}(p_{\text{pre}} - p_{\text{next}}, rt[s/p - 1 \to 0]);\)
37: \(rt[s/p - 1 \to 0] = \text{receive}(p_{\text{pre}} - p_{\text{next}});\)
38: \(\text{store}(rt[s/p - 1 \to 0] \to T \text{ in the correlated location}; \{\text{share results}\}\)

<table>
<thead>
<tr>
<th>Table 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Meanings of Symbols</strong></td>
</tr>
<tr>
<td>(w)</td>
</tr>
<tr>
<td>(s)</td>
</tr>
<tr>
<td>(p)</td>
</tr>
<tr>
<td>(P)</td>
</tr>
<tr>
<td>(q)</td>
</tr>
</tbody>
</table>
To get balanced task partitioning, TBs, which are assigned to \( p \) cores in a circular fashion. There are many ways to distribute the result has been generated but not shared: each core has one piece of it. There are many ways to distribute the result through every core one by one (\( \text{line 32 to line 38 in Algorithm 2} \)).

Montgomery multiplication, we group every columns of boxes to form different TBs. In total, there are \( 2 \cdot s/q \) TBs, which are assigned to \( p \) cores in a circular fashion. To get balanced task partitioning, \( s \) should be divisible by \( p \cdot q \). The algorithm is shown in Algorithm 2. After all TBs are processed (by line 31 in Algorithm 2), the final result has been generated but not shared: each core has one piece of it. There are many ways to distribute the result to every core. Line 32 to line 38 in Algorithm 2 implements a basic one: every piece of the result goes through every core one by one (\( p - 1 \) steps in total).

4 Analysis

This section analyzes the influence of inter-core communication latency on the overall performance of pSHS. Given a system and the application, unlike other parameters such as \( p \), \( s \), and \( \text{CTU} \) which have only a few possibilities, the inter-core communication latency \( \text{TTU} \) is usually more dynamic. This is because it is closely related to the status of the traffic in the communication network. Therefore, while the relationships between the performance and other parameters can be easily profiled with only a few trials by running the program on the system, the \( \text{TTU}'s \) influence needs to be formulated in a mathematical way. Actually, analyzing the influence of \( \text{TTU} \) is pretty complicated. To make it easier to understand, we first use an example to explain our analysis method. After that, a general quantitative analysis is presented. In the end, we are able to formulate the relationship between \( \text{TTU} \) and \( \text{pSHS}'s \) performance when \( \text{TTU} \) is at a typical range, which shows that \( \text{TTU}'s \) influence on \( \text{pSHS}'s \) performance is usually acceptable.

4.1 Analysis on an example

We use the same example mentioned in last section, in which a 6-word MM is processed by 3 cores. The analysis is performed in 3 steps starting from an ideal case and then approaching the answer to the above questions by considering practical and stricter conditions.

4.1.1 First step

Assume that \( \text{TTU} = 0 \). We first consider an ideal case where message communication latency is 0. According to time, we draw the parallel execution based on pSHS in Figure 4(a). The x-axis represents time. Six TBs with their boxes are located in six rows. The length of every box represents its execution time: CTU. The x-position of each box is decided by its starting time. Arrows show the message transfers. The length of the projection of one arrow on the x-axis indicates the longest time allowed for that message transfer, named \( \text{allowance time} \) (AT). We find that all ATs are no smaller than 0. Under the assumption that \( \text{TTU} = 0 \), all messages arrive at their destinations before the cores try to 'receive' them. This leads to a full parallelization of calculation. The execution time of Algorithm 2 (\( T_{\text{pSHS}} \)) is \( 24 \cdot \text{CTU} \).

4.1.2 Second step

Assume that \( 0 < \text{TTU} < \text{CTU} \) and only one transfer exists between two cores. In this case, all cores receiving message with \( \text{AT} = 0 \) have to wait for the messages. In Figure 4(b), the black areas in TB[4] and TB[5] represent the time when P1 and P2 wait for the messages. After that, P2 needs to notify P0 the end of process. Finally, another 2 steps of transfers are used to distribute the final result. Therefore, \( T_{\text{pSHS}} \) is \( 24 \cdot \text{CTU} + 8 \cdot \text{TTU} \), shown in Figure 4(b).

4.1.3 Third step

Assume that \( \text{TTU} > \text{CTU} \) and only one transfer exists between two cores. In this case, besides P1 and P2, also P0 has to wait for messages when processing TB[3], shown
in Figure 4(c). Thus, \( T_{pSHS} \) will increase even more and even faster than the second step. As TTU increases further, black areas will eventually appear in TB[1] and TB[2], which results in \( T_{pSHS} \)'s fastest increasing speed against TTU.

According to the above analysis, we find that \( T_{pSHS} \) is a piecewise function of TTU. As TTU increases, \( T_{pSHS} \)'s slope also increases from piece to piece.

4.2 Quantitative analysis on general cases

With a similar analysis method, in this section, we perform analysis on general cases. We define 3 terms: available time, request time and allowance time, all of which are related to the inter-core communications. Suppose core \( P_i \) needs to transfer a data \( D \) to core \( P_{i+1} \). available time of \( D \) is the time when \( P_i \) sends \( D \) to the inter-core network; request time of \( D \) is the time when \( P_{i+1} \) tries to receive \( D \) from the network; allowance time = (request time - available time)/(number of communications). Allowance time indicates the longest average transfer time for one communication that guarantees that \( P_{i+1} \) does not have to wait for the message. Table 2 summarizes the terms that will be used in the following analysis.

With the similar method as in Section 4.1, we perform three steps of analysis as follows.

4.2.1 First step: \( TTU = 0 \)

When \( TTU = 0 \), a core has to be stalled if the message received by it has a negative allowance time. During processing a TB, one core needs to receive several words of \( m \) and two words of \( c \). For example, \( P_i \) needs to receive \( m_0 \) and \( m_1 \) when processing TB[1] in Figure 3. For different \( ms \) transferred from TB[i] to TB[i+1], their allowance time could be different. We define \( t_m[i] \) as the smallest allowance time of messages transferred from TB[i] to TB[i+1]. If TTU is smaller than \( t_m[i] \), the processing on TB[i+1] does not need to wait for the messages sent from the process that handles TB[i]. Similarly, we use \( t_c[i] \) to represent the smallest allowance time of \( c \) transferred from TB[i] to TB[i+1]. Regardless of the data dependencies, the general expression of \( t_m[i] \) is as follows, where \( j = 1, 2, 3, \ldots \).

\[
\frac{t_m[i]}{CTU} = \begin{cases} 
2i^2, & i < s/p - 1, \ i \neq j/p - 1 \\
2(\frac{s}{p} + 1)q^2 + q, & i < s/p - 1, \ i = j/p - 1 \\
2(\frac{s}{p} - 2)q^2 - \frac{s}{q} & i = s/p - 1 \\
2(\frac{s}{p} - 1)q^2, & s/p - 1 \leq i < \frac{s}{q} + p - 2 
\end{cases}
\]

Once \( i \geq s/q + p - 2 \), there is no need to transfer \( m \). With the same method, we can also obtain \( t_c[i] \) as follows, where \( j = 1, 2, 3, \ldots \).

\[
t_c[i] = \begin{cases} 
2i^2, & i < s/p - 1, \ i \neq j/p - 1 \\
2(\frac{s}{p} + 1)q^2 + q, & i < s/p - 1, \ i = j/p - 1 \\
2(\frac{s}{p} - 2)q^2 - \frac{s}{q} & i = s/p - 1 \\
2(\frac{s}{p} - 1)q^2, & s/p - 1 \leq i < \frac{s}{q} + p - 2 
\end{cases}
\]
We draw \( t_m[i] \) and \( t_c[i] \) in Figure 5 as an example with \( q = s/p \). According to the general expressions of \( t_m[i] \) and \( t_c[i] \), when \( i < p, t_m[i] \) and \( t_c[i] \) are both larger than 0. As \( i \) increases, both \( t_m[i] \) and \( t_c[i] \) decrease. For the last \( p - 1 \) TBs, their \( t_m[i] \) and \( t_c[i] \) become 0. Even if we consider the data dependencies, when \( TTU = 0 \), pSHS still achieves a full parallelization during the calculation.

### 4.2.2 Second step: \( TTU > 0 \) and only one communication exist between two cores

In this case, the full parallelization is not realizable since the last \( p - 1 \) TBs will definitely have to wait for the messages \( (t_c[i] = t_m[i] = 0, p \leq i \leq 2p - 2) \). All the other TBs \( (t_m[i], t_c[i] > 0, 0 \leq i \leq p - 1) \) could be uninfluenced by the communication delay, when \( TTU \) is smaller than a certain limit (similar to the second step in Section 4.1). We define this communication delay limit as communication latency tolerance (CLT). CLT guarantees that the communication latency has the smallest influence to pSHS’s execution time. As discussed in Section 4.1, pSHS’s execution time is a piecewise function of \( TTU \). Intuitively, CLT indicates the largest \( TTU \) of the first piece. The following discussion discovers CLT and the execution time of pSHS.

**Discovering CLT**

CLT means the largest \( TTU \) that guarantees TBs with non-zero \( t_m[i] \) and \( t_c[i] \) not delayed by the communication delay. We first consider the case when \( q = s/p \) (the largest \( q \) we can choose).

For TBs with non-zero \( t_m[i] \) and \( t_c[i] \), the available time and request time of each data transfer from TB\([i]\) to TB\([i+1]\) can be illustrated in Figure 6. The horizontal axis represents time. A bunch of horizontal lines are used to represent data transfers from TB\([i]\) to TB\([i+1]\) with the starting point of a line as the available time and the end point as the request time. We categorize the lines into three groups. In GROUP1, TB\([i]\) sends \((p - 2) \times q \) words of \( m \) to TB\([i + 1]\), which are received from the previous \( p - 2 \) cores. In GROUP2, TB\([i]\) generates \( q \) words of \( m \), and sends them to TB\([i+1]\). In GROUP3, TB\([i]\) generates 2 words of \( c \) and sends them to TB\([i+1]\).

For simplicity, we divide the time domain into 3 parts corresponding to the three groups of lines: from A to B (\( AB \)), from B to C (\( BC \)), and from C to D (\( CD \)) in Figure 6. Accordingly, we get \( AB = ((p - 2)q^2) \times CTU \), \( BC = ((q^2 + q) / 2) \times CTU \), \( CD = (q^2 - q) \times CTU \) when \( i = q \) or \( CD = (q^2 + 2q - 3) \times CTU \) when \( i < q - 1 \).

For \( AB \), if \( TTU \leq q \times CTU \), the next TB does not have to wait for the messages in GROUP1. For messages in GROUP2 and GROUP3, if the communication network can finish \( q + 2 \) communications within time \((3q^2 - q) \times CTU / 2\), the next TB can be processed without waiting for messages in GROUP2 and GROUP3. Therefore, if the \( TTU \) is smaller than \( \min(q \times CTU, (3q^2 - q) \times CTU / 2) \), the next TB does not have to wait for any message sent from the current TB, and this is the value of CLT. We approximate CLT with Equation 1.

\[
CLT = \min(q \times CTU, \frac{(3q^2 - q)}{2(q + 2)} \times CTU)
\]

**Discovering execution time**

When \( 0 < TTU < CLT \), the execution time changes as \( TTU \) changes because TB\([i]\) with \( t_m[i - 1] \) or \( t_c[i - 1] \) being 0 are delayed, which corresponds to TB\([i]\) (\( i > p \)) in Figure 4(b). In this case, core \( P_i \) is stalled by \( 2(s+i-1)TTU \). The overall calculation is delayed by \( 2q \times TTU \) (\( P_{p-1} \) sends two words of \( c \) to \( P_0 \) as the finishing signal). In addition, to share the final result, \( p - 1 \) steps of data transfer are needed. Therefore, the total delay is \((3p - 1)TTU\).

When \( TTU > CLT \), the processing cores also have to wait for messages when processing other TB\([i]\) (\( i \leq p \)). Therefore, \( T_{pSHS} \) increases more quickly than the previous case. In detail, since the interval between two consecutive \( m \) messages are \( q \times CTU \) which approximately equals to \( CLT \), once \( TTU > CLT \), every TB except TB\([0]\) will be delayed by the communication latency. The additional delay on TB\([i]\) (\( i \leq p \)) depends on \( q \) \( m \) messages and \( 2c \) messages. So the delay of the first half MM increases \( p \times (q + 2) \) times faster than \( TTU \). When \( i > p \), the increase of the execution time is exactly the same as the previous case when \( TTU < CLT \), which is \((3p - 1) \) times faster than \( TTU \). After adding these two parts together, we can see that, when \( TTU > CLT \), the execution time increases \( pq + 5p - 1 \) times faster than \( TTU \).

Considering SOS scheme is used in every TB, the time cost of pSHS can be represented by Equation 2, where \( PO \) means pSHS’s parallel overhead due to the increased complexity when compared with SOS.

\[
T_{pSHS} = \begin{cases} \frac{T_{pSHS}}{p} + (3p - 1) \times TTU + PO, & TTU \leq CLT \\ (pq + 5p - 1) \times (CLT - CLT) + T_{pSHS}(CLT), & TTU > CLT \end{cases}
\]

Following the same method, we can also analyze the cases when \( q < s/p \). The analysis results are similar.

### 4.3 Analysis conclusion

This section evaluates pSHS’s typical performance according to the formulas obtained above. In addition, we also discuss how to choose optimal values for certain parameters to achieve the best performance.
4.3.1 Optimal $q$ and $p$

Usually, given a platform and the application, most of the parameters are fixed, such as $s$ and $CTU$. Parameters under programmers’ control are $q$ and $p$. Thus, we need to figure out the optimal number of columns of boxes that are grouped into one TB and the optimal number of cores that should be used to map pSHS.

From Equation 1, we can see that the largest $q$ leads to the largest $CLT$. In addition, a smaller $q$ requires more iterations of the i-loop in Algorithm 2, and hence parallel overhead is larger. Therefore, we should always choose the largest possible number for $q$: $(s/p)$.

The relationship between $T_{pSHS}$ and $TTU$ is shown by Equation 2, when $TTU$ is smaller than the communication latency tolerance. When $TTU$ is larger than the tolerance, time cost of pSHS increases more quickly. Therefore, when $TTU$ is small, we expect that a large number of cores will provide a better overall performance for pSHS. However, as $TTU$ increase, larger $p$ indicates higher increasing speed $(3p-1)$ of the execution time. Therefore, it is possible that larger $p$ does not always give better performance.

To find the optimal value of $p$ purely based on Equation 2, we need to further formulate the term $PO$ (the parallel overhead), which is related to the increased complexity of pSHS when compared with the sequential implementation. However, the term is highly system-dependent. Fortunately, we find another simpler way which can fulfill the same purpose. Since $PO$ is not related to $TTU$, it can be measured on given a specific system. Due to the limited possibilities of other parameters, $PO$ also have only a few possible values. After fixing $PO$, we can use Equation 2 to compare different choices of $p$ under different values of $TTU$ and finally obtain the optimal $p$ for a given $TTU$. Both of the above conclusions will be demonstrated in Section 6.

4.3.2 Evaluation of pSHS’s typical performance

We can see that $T_{pSHS}$ is a piecewise function of $TTU$; when $TTU < CLT$, $T_{pSHS}$ stays on the first piece with the lowest increasing slope: $3p-1$; when $TTU > CLT$, $T_{pSHS}$ moves to another piece with higher increasing slope.

To process a $s$-word MM, $2s^2$ boxes should be processed in total. The execution time for a sequential program should be approximately $2s^2 * CTU$. The optimal execution time of pSHS should be $2s^2 * CTU/p$. As $TTU$ increases from $0$, $T_{pSHS}$’s increasing rate, when compared with the optimal situation, is $(3p-1)p/2s^2 * CTU$. In each box, 4 additions, 1 multiplication, 2 memory loads, 1 memory store, and 1 conditional jump are processed. Usually $CTU$’s typical value is around 20 clock cycles. Suppose the application is 64-word MM (e.g. 2048-bit MM processed by a 32-bit 8-core processor), then the increasing rate equals to 0.1%. Even if the on-chip communication delay reaches $CLT$ ($q * CTU = 160$ clock cycles), $T_{pSHS}$ increases by 16%, when compared with the optimal case. The performance is degraded to $1/1.16 = 86\%$. Moreover, considering the influence of parallel_overhead, the actual degradation is even less. Therefore, when $TTU < CLT$, the influence of $TTU$ on $T_{pSHS}$ is acceptable. Plus, 160 clock cycles of delay also does not present a very high requirement to on-chip communications. Moreover, larger $CTU$, smaller word length of the processor, and longer operands of MM will all increase $CLT$ and decrease $T_{pSHS}$’s increasing speed.

5 Implementing RSA with pSHS

In order to verify pSHS’s effects on accelerating public-key cryptography, we implement a parallel RSA with pSHS. A small modification is made to pSHS in Algorithm 2 by adding the final subtraction. RSA’s encryption and decryption are modular exponentiation which is shown in Algorithm 3.
As we can see in Algorithm 3, modular exponentiation is mainly a set of MMs. Line 1 to line 3 do the preparation and do not cost much time. Therefore, the parallel modular exponentiation’s speedup should be very close to pSHS’s.

6 Experimental Results

6.1 Experimental Setup

Experiments were built on a Xilinx Virtex 5 FPGA, where we implemented three multi-core prototypes with 2, 4, and 8 MicroBlaze cores (w=32, running at 100MHz) respectively. All cores are connected in a ring network, as shown with a 4-core example in Figure 7. Each core works independently with its own local memory. Communications between neighboring cores are implemented with Fast Simplex Link (FSL) buses. We modify FSL and make its delay programmable to emulate different communication latencies. Since the on-chip communication is not likely to have errors, the communication does not include a feedback from the receiver to the sender to verify the message is correct. A timer is attached to MicroBlaze 0 to measure the execution time. All cores and the timer are synchronized at the starting time. After every core finishes its job, MicroBlaze 0 reads the clock cycle counts from the timer, which is used as the execution time.

The FPGA with soft-cores is just for prototyping. It represents a category of multi-core architectures where different cores run independently and inter-core communication is done with a message-passing network. The programmable communication delay enables our prototype to represent different topologies of the network. The delay in the prototype indicates the worst case of the communication in all topologies. Therefore, we believe the results we obtained from the FPGA prototype also holds valid when applied to some existing and emerging multi-core architectures, for example the tiled processors [24, 25, 26, 27, 28, 32].

Algorithm 3 parallel modular exponentiation based on pSHS

Require: An s-word modulus \( N = (n_{s-1}, n_{s-2}, ..., n_1, n_0) \), base \( A = (a_{s-1}, a_{s-2}, ..., a_1, a_0) \), with \( 0 < A < N \), and exponent \( E = (e_{s-1}, e_{s-2}, ..., e_1, e_0) \) (\( E > 0 \)). \( R = 2^s (\text{mod} N) \), \( n = w \times s \). pSHS\((A, B, N, i)\) comes from Algorithm 2, which calculates \( A 	imes B 	imes R^{-1} (\text{mod} N) \) in core \( P_i \).
Ensure: \( T = (t_{s-1}, t_{s-2}, ..., t_1, t_0) = A^E (\text{mod} N) \). \( T \) is stored locally for each core.
The complete modular exponentiation consists of \( p \) copies of the following program. On core \( P_i \), execute:

1. \( pid = P_i \);
2. \( A = A \times R \ (\text{mod} N) \);
3. \( T = R \ (\text{mod} N) \);
4. for \( i = 0 \) to \( n - 1 \) do
   5. if \( e_i = 1 \) then
   6. \( T = \text{pSHS}(A, T, N, pid) \);
   7. \( A = \text{pSHS}(A, A, N, pid) \);
   8. \( T = \text{pSHS}(T, 1, N, pid) \);

Fig. 7. 4-core architecture with distributed local memory and FSL based message passing connection.

Fig. 8. pSHS’s execution time of processing 1024-bit Montgomery multiplication with 4 cores (\( s = 32, p = 4 \)): larger \( q \) has better performance and larger communication delay tolerance. Execution time increases faster for smaller \( q \) as TTU increases.

6.2 Experimental Results

In each platform, we implemented 512 (\( s = 16 \)), 1024 (\( s = 32 \)), and 2048 (\( s = 64 \)) bit-long MMs and modular exponentiations. We programmed each of these in C according to Algorithm 2 and Algorithm 3. The compiler’s optimization level is 2.

6.2.1 Finding the optimal number of columns in one TB

In every MM we tested, \( q = s/p \) always leads to the best performance. We show the result with \( s = 32, p = 4 \) in Figure 8.

When TTU is small, the influence of communication delay is limited. However, since a smaller \( q \) requires more iterations of the i-loop in Algorithm 2, the performance is worse. Moreover, smaller \( q \) also leads to smaller delay tolerance, which means that its performance deteriorates more quickly as TTU increases. These two phenomena can both be observed in Figure 8. Therefore, our conclusion in Section 4.3 that \( q \) should be \( s/p \) has been
TABLE 3  
<table>
<thead>
<tr>
<th>operand</th>
<th>512 bits</th>
<th>1024 bits</th>
<th>2048 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>execution time (cycles)</td>
<td>13953</td>
<td>53905</td>
<td>212145</td>
</tr>
</tbody>
</table>

| TABLE 4  
| pSHS’s Execution Time (et, cycles) & Speedup (sp) |
|----------|----------|----------|
| p=2  | p=4  | p=8  |

<table>
<thead>
<tr>
<th>TTU</th>
<th>et</th>
<th>sp</th>
<th>sp</th>
<th>sp</th>
<th>sp</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>32</td>
<td>8804</td>
<td>9144</td>
<td>10144</td>
<td>14563</td>
</tr>
<tr>
<td></td>
<td>1.58</td>
<td>1.52</td>
<td>1.38</td>
<td>0.96</td>
<td>0.71</td>
</tr>
<tr>
<td>1024</td>
<td>29382</td>
<td>29856</td>
<td>30356</td>
<td>31856</td>
<td>31474</td>
</tr>
<tr>
<td></td>
<td>1.83</td>
<td>1.81</td>
<td>1.78</td>
<td>1.75</td>
<td>1.72</td>
</tr>
<tr>
<td>2048</td>
<td>107228</td>
<td>107568</td>
<td>108568</td>
<td>109568</td>
<td>11753</td>
</tr>
<tr>
<td></td>
<td>1.98</td>
<td>1.97</td>
<td>1.96</td>
<td>1.94</td>
<td>1.92</td>
</tr>
<tr>
<td>p=8</td>
<td>32</td>
<td>4364</td>
<td>4752</td>
<td>6333</td>
<td>9003</td>
</tr>
<tr>
<td></td>
<td>2.46</td>
<td>2.30</td>
<td>2.11</td>
<td>1.95</td>
<td>1.60</td>
</tr>
<tr>
<td>1024</td>
<td>16716</td>
<td>17090</td>
<td>18190</td>
<td>19334</td>
<td>22609</td>
</tr>
<tr>
<td></td>
<td>3.22</td>
<td>3.15</td>
<td>3.06</td>
<td>2.96</td>
<td>2.88</td>
</tr>
<tr>
<td>2048</td>
<td>56777</td>
<td>57151</td>
<td>58251</td>
<td>59351</td>
<td>60451</td>
</tr>
<tr>
<td></td>
<td>3.74</td>
<td>3.71</td>
<td>3.68</td>
<td>3.64</td>
<td>3.61</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TTU</th>
<th>et</th>
<th>sp</th>
<th>sp</th>
<th>sp</th>
<th>sp</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>32</td>
<td>4364</td>
<td>4752</td>
<td>6333</td>
<td>9003</td>
</tr>
<tr>
<td></td>
<td>2.94</td>
<td>2.62</td>
<td>2.20</td>
<td>1.83</td>
<td>1.60</td>
</tr>
<tr>
<td>1024</td>
<td>10762</td>
<td>11150</td>
<td>12300</td>
<td>13450</td>
<td>15486</td>
</tr>
<tr>
<td></td>
<td>5.01</td>
<td>4.83</td>
<td>4.60</td>
<td>4.38</td>
<td>4.19</td>
</tr>
<tr>
<td>2048</td>
<td>32470</td>
<td>32858</td>
<td>34008</td>
<td>35158</td>
<td>36308</td>
</tr>
<tr>
<td></td>
<td>3.20</td>
<td>2.94</td>
<td>2.62</td>
<td>2.20</td>
<td>1.83</td>
</tr>
</tbody>
</table>

11. Therefore, the relationship between $T_{pSHS}$ and TTU shown in Equation 2 has been demonstrated to be true for the 4-core architecture. With the same method, we find that it is also true for 2- and 8-core architectures.

When TTU is small, 4-core based pSHS’s speedup can be as high as 3.74, 3.22, and 2.46 for three operand lengths. Even when TTU = 550 cycles, the speedup of 2048-bit pSHS is still above 3.5. Based on 2 cores, the speedup of 2048-, 1024-, and 512-bit pSHS can be as good as 1.98, 1.83, and 1.58 respectively. Based on 8 processors, they can be 6.53, 5.01, and 3.20. Generally speaking, pSHS provides a good speedup.

Modular exponentiation’s execution time highly depends on the number of ‘1’s in the exponent. In our experiment, the exponents were randomly generated. So the probability of each bit to be ‘1’ is close to 0.5. For comparison, we use the same inputs ($A$, $B$, and $N$) for both sequential modular exponentiation (with SOS) and parallel modular modular exponentiation (with pSHS). The results of the sequential and parallel versions are listed in Table 5 and Table 6 respectively.

After integrating pSHS to the modular exponentiation, the modular exponentiation’s speedup is very close to pSHS’s speedup, with a small drop. This lower speedup comes from the final subtraction in the modified pSHS and the preparation process, which are not parallelized. Therefore, our Conclusion 3 in Section 5 has also been demonstrated.

6.2.3 Finding the optimal number of cores

From Table 4, we realize that if the operand width ($s$) and the number of cores ($p$) are fixed, $T_{pSHS}$ is only determined by TTU with the same slope shown in Equation 2. This demonstrates that parallel overhead is irrelevant to TTU. As a programmer, to obtain the optimal value of $p$, the first step is to fix ‘parallel overhead’. For example, if the application is 1024-bit MM, three measurements of $T_{pSHS}$ on the dual-, quad-, and octo-core platforms under a known TTU ($TTU < CLT$) would reveal parallel overhead = 2404, 3064, and 3449.
TABLE 5
Execution Time of sequential modular exponentiation based on SOS

<table>
<thead>
<tr>
<th>operand</th>
<th>execution time (cycles)</th>
<th>512 bits</th>
<th>1024 bits</th>
<th>2048 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>p=2</td>
<td>7006</td>
<td>10942151</td>
<td>85568236</td>
<td>646488715</td>
</tr>
<tr>
<td>p=4</td>
<td>46737</td>
<td>47270</td>
<td>48840</td>
<td>50410</td>
</tr>
<tr>
<td>p=8</td>
<td>32815</td>
<td>32919</td>
<td>33226</td>
<td>33532</td>
</tr>
</tbody>
</table>

TABLE 6
Execution Time (et, 10000*cycles) & Speedup (sp) of parallel modular exponentiation based on pSHS

<table>
<thead>
<tr>
<th>TTU</th>
<th>et</th>
<th>sp</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 10. 1024-bit pSHS’s execution time on 2-, 4-, and 8-core platforms based on calculation according to Equation 2.

respectively in three platforms, according to Equation 2. Based on that, we are able to calculate the values of TTU that result in cross points among $T_{pSHS}$’s different curves corresponding to different $p$ values. Suppose $T_{pSHS}$ of dual-core and quad-core platforms are equal when $TTU = TTU_{cross}$, then if $TTU < TTU_{cross}$ quad-core platform performs better, when $TTU > TTU_{cross}$ dual-core platform performs better. We draw the curves in Figure 10 based Equation 2. The cross point between dual- and quad-core platforms occurs when $TTU = 600$; the cross point between quad- and octo-core platforms occurs when $TTU = 200$. From the experiment results in Table 4, we can see that these two crossing points occurs around $TTU = 550$ and $TTU = 225$. In comparison, we can see that the calculated results and the experimental results are very close. Therefore, our method to find the optimal number of cores is correct.

6.3 Discussion
The multicore platforms we built are very close to the worst-case analysis model. Given better conditions, such as better topology, larger capacity of communication channels, the performance and communication latency tolerance can be improved further. However, even so, we already see a good speedup. Furthermore, using several hundreds of cycles to transfer one message between neighboring cores is not a difficult requirement for current on-chip multicores systems. Therefore, pSHS provides a good performance, a good portability and a good stability.

We find that the efficiency of pSHS is closely related to the number of words in the multiplication operand ($s$). Larger $s$ leads to higher efficiency and vice versa. Based on the results, we find that pSHS has a very high efficiency when used to accelerate RSA and DSA on 32-bit systems. In addition, moving from Montgomery multiplication to modular exponentiation does not require too much additional operations. Therefore, it is reasonable to expect a good performance after integrating pSHS to RSA and DSA. Because of shorter operands, based on 32-bit systems, ECC may not benefit as much of pSHS as RSA and DSA. However, in many low-end implementations where the word length of the cores is 8 bits [33] [34], pSHS can still be a good candidate for acceleration.

Ideally, pSHS requires $s$ to be dividable by $p\times q$. In some cases, this requirement cannot be satisfied. An easy solution is to extend the operands with 0 until the requirement is satisfied. Also, as the number of cores ($p$) increases to be equal to $s$, we can hardly gain additional speedup by adding more cores for computation. Even so, pSHS already shows a good scalability. For 1024-bit RSA, this limit number of $p$ is 64 for 32-bit cores and 256 for 8-bit cores.

Compared with the parallelization obtained by concurrently performing multiple encryptions, whose ideal speedup could be linear, pSHS trades some throughput for much lower latency. Future work will investigate the combination of these two methods to find suitable tradeoffs between throughput and latency for different applications.

7 Conclusion
In this paper, we propose pSHS as a parallel programming scheme for Montgomery multiplication based on multicore systems. Both analysis and experiments on real multicore prototypes show that pSHS provides a good
speedup, large communication latency tolerance, good portability and good scalability. These features make pSHS a good parallel software solution for RSA, DSA, and ECC on multicore systems.

8 Acknowledgments
This research has been supported in part by National Science Foundation (NSF) Grant No. 0644070.

References
Zhimin Chen received the BS degree in Electrical and Information Engineering from Huazhong University of Science and Technology, China, in 2004, and the MS degree in Electrical Engineering from Shanghai Jiao Tong University, China, in 2007. He is currently a PhD student in Computer Engineering at Virginia Tech. His research interests include applied cryptography and computer architecture, in particular, high-performance and trustworthy secure systems.

Patrick Schaumont is Assistant Professor in Computer Engineering at Virginia Tech. He received the PhD degree in Electrical Engineering from UCLA (2004), and the MS degree in Computer Science from Rijksuniversiteit Ghent, Belgium (1990). His research interests include design of, and design methodologies for, secure embedded systems. He has served on the program committee of international conferences in this field including CHES, DATE, DAC, IEEE HOST and IEEE MEMOCODE, and as guest editor for IEEE Design and Test Magazine, ACM Transactions on Reconfigurable Technology and Systems, and IEEE Transactions on Computer-Aided Design of Integrated Circuits. He is a senior member of the IEEE.