

pSHS: A Scalable Parallel Software Implementation of Montgomery Multiplication for Multicore Systems

Zhimin Chen

Department of Electrical and Computer Engineering
Virginia Tech
Blacksburg, Virginia 26061
Email: chenzm@vt.edu

Patrick Schaumont

Department of Electrical and Computer Engineering
Virginia Tech
Blacksburg, Virginia 26061
Email: schaum@vt.edu

Abstract—Parallel programming techniques have become one of the great challenges in the transition from single-core to multicore architectures. In this paper, we investigate the parallelization of the Montgomery multiplication, a very common and time-consuming primitive in public-key cryptography. A scalable parallel programming scheme, called pSHS, is presented to map the Montgomery multiplication to a general multicore architecture. The pSHS scheme offers a considerable speedup. Based on 2-, 4-, and 8-core systems, the speedup of a parallelized 2048-bit Montgomery multiplication is 1.98, 3.74, and 6.53, respectively. pSHS delivers stable performance, high portability, high throughput and low latency over different multicore systems. These make pSHS a good candidate for public-key software implementations, including RSA, DSA, and ECC, based on general multicore platforms. We present a detailed analysis of pSHS, and verify it on dual-core, quad-core and eight-core prototypes.

I. INTRODUCTION

For higher performance, processor vendors have been adding momentum to the single-core-to-multicore transition push. However, unlike increasing the clock frequency of a single core, turning to multicore does not give a free ride to sequential software. Therefore, parallel programming has become a great challenge for software engineers [1]. In this paper, we take the challenge and look into the problem on how to apply parallel programming to public-key cryptography, the most time-consuming category of cryptographic algorithms.

Previous work falls into three categories. The first is the HW/SW codesign method [2] [3], in which customized parallel architectures are built for efficient implementation. The second category makes use of the special parallel processing features in commodity processors, such as multiple datapaths in DSP [4], or SSE2 in Pentium 4 [5]. The third executes up to thousands of encryptions or decryptions in parallel by making use of the large number of processing units in a Graphics Processing Unit (GPU) [6] [7] [8].

All of these efforts achieved their successes, but they also have limitations. *Lack of portability* is a limitation for the first and the second categories, since they are strictly constrained by a specific computing platform. When mapped to other systems, they may not work well because of larger communication

latency, different topology, or lack of some special parallel processing resources. *Long latency* is a problem for the third category. This is because parallelism is not achieved by parallelizing one encryption or decryption. As the complexity of cryptography increases, these methods will eventually end up with an unbearable latency. For example, one 2048-bit modular exponentiation on a high-end GPU platform has a minimum latency of almost one minute [6]. This is not convenient for users. The situation will be even worse as the cryptographic complexity increases further. Due to the above problems, a parallel programming scheme for public-key cryptography based on general multicore systems is still in need.

In this paper, we address both of the above problems by proposing a scalable parallel programming scheme for the Montgomery multiplication [9]. Our scheme is called parallel Separated Hybrid Scanning (pSHS). pSHS achieves a high parallelism and a high scalability. It 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 Montgomery multiplication implemented on 32-bit embedded cores has speedups of 1.98, 3.74, and 6.53 based on 2-, 4-, and 8-core architectures respectively. The efficiencies per core are as high as 0.99, 0.94, and 0.82 respectively. Compared with the existing related work, pSHS has 3 advantages. 1) pSHS has a more stable performance over different multicore architectures. This is because it can accommodate up to several hundred clock cycles of communication latency without significant impact to the overall performance. 2) pSHS has a better portability since it has no requirements to the topology nor the special parallel processing resources. 3) pSHS supplies both high throughput and low latency to public-key cryptography, including RSA [10], DSA [11], and ECC [12], by accelerating one single Montgomery multiplication. We have built real multicore prototypes using dual-core, quad-core and eight-core systems to demonstrate our arguments.

II. MONTGOMERY MULTIPLICATION

Montgomery multiplication is one of the most prevalent ways to implement modular multiplications and exponen-

Algorithm 1 Montgomery multiplication (SOS scheme) [13]

Require: An s -word modulus $N = (n_{s-1}, n_{s-2}, \dots, n_1, n_0)$, two operands $A = (a_{s-1}, a_{s-2}, \dots, a_1, a_0)$, and $B = (b_{s-1}, b_{s-2}, \dots, b_1, b_0)$ with $A, B < N$, and the constant $n' = -n_0^{-1} \bmod 2^w$. w is the word length of a system (usually 8, 32, or 64). $T = (t_{2s}, t_{2s-1}, \dots, t_1, t_0)$ is an intermediate variable.

Ensure: $U = (u_s, u_{s-1}, \dots, u_1, u_0) = A * B * 2^{-n} \bmod N$. $n = w * s$.

```

1:  $T = 0$ ;
2: for  $i = 0$  to  $s - 1$  do
3:    $C = 0$ ;
4:   for  $j = 0$  to  $s - 1$  do
5:      $(C, S) = t[i + j] + a[j] * b[i] + C$ ;
6:      $t[i + j] = S$ ;
7:      $t[i + s] = C$ ;
8:   for  $i = 0$  to  $s - 1$  do
9:      $C = 0$ ;
10:     $m = t[i] * n' \bmod 2^w$ ;
11:    for  $j = 0$  to  $s - 1$  do
12:       $(c, d) = t[i + j] + m * n[j] + c$ ;
13:       $t[i + j] = S$ ;
14:       $ADD(t[i + s], C)$ ;
15:    for  $j = 0$  to  $s$  do
16:       $u[j] = t[j + s]$ ;
17:    if  $U > N$  then
18:       $U = U - N$ ;

```

tiations. One of its sequential implementation schemes is shown in Algorithm 1 [13]. The most time-consuming part of Algorithm 1 is from line 2 to line 14. Our research only focuses on this part and ignores the final subtraction in line 15 and 16, since it does not affect the overall performance too much.

After profiling Algorithm 1, we find two hot blocks: the two inner loops between line 5 and line 6 containing $a_j * b_i$ and between line 12 and line 13 containing $n_j * m_i$. To visualize the analysis, we use a white box to represent one iteration of the first loop and a shaded box for the second loop, shown in Figure 1(a). The inputs of a box include t_{in} and c_{in} . m_{in} is a third input to a shaded box. Every box generates two words of output with the higher part in c_{out} and the lower one in t_{out} . Figure 1(b) shows the data flow and data dependency of a 4-word Montgomery multiplication. Boxes are located according to their index (i, j) . In total, there are 8 columns and 8 rows of boxes. Adding the partial results of all the boxes, we obtain the Montgomery product by keeping the upper half of the final result. In Figure 1(b), all the vertical connections represent dependencies based on t , while the horizontal connections represent dependencies based on c and m . All of the shaded boxes in a row share the same m , which is generated right before the first box in that row. Since the operations inside both the white and the shaded boxes are similar, we assume that they take the same amount of time, and define it as *calculation time unit* (CTU). Besides the boxes, other operations, for example the generation of m in line 10, are simpler than the operations in one box. We approximate the time cost of them to be 0.

III. MULTI-CORE MODEL

To make our programming scheme portable, we base our parallel scheme on a general multicore model as follows. In the multicore architecture, processing cores work independently

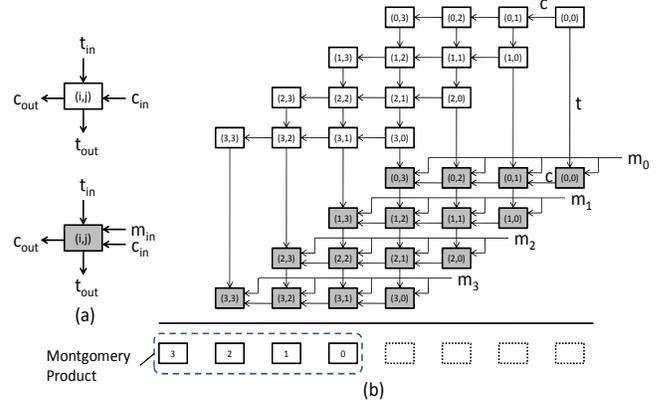


Fig. 1. (a) Analysis model: boxes; (b) Data flow of Montgomery multiplication

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.

IV. PARALLEL SEPARATED HYBRID SCANNING

A. 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 2, we show the example of a 6-word Montgomery multiplication 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 [13]. 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*

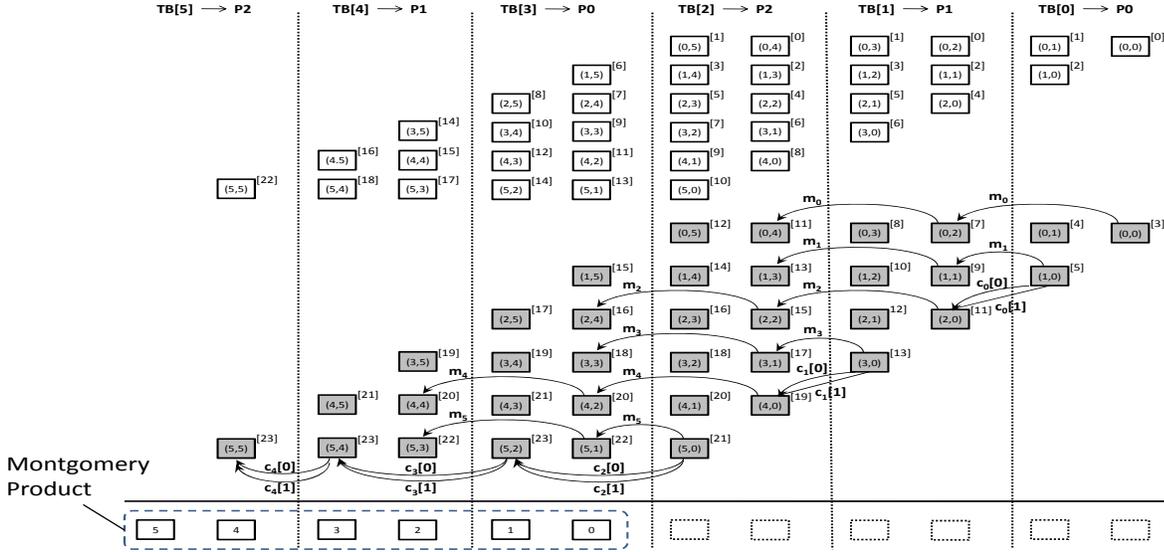


Fig. 2. An example of pSHS ($s = 6, p = 3, q = 2$)

as the end. All data communications between different TBs are shown as arrows in Figure 2. 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 each 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 I. In an s -word Montgomery multiplication, we group every q columns of boxes to form different TBs. In total, there are $2 * s/q$ TBs, which are assigned to p cores in a circular fashion. To get balanced task partitioning, s should be divisible by $p * q$. 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).

B. Analysis

The analysis in this section answers two questions. First, how many columns of boxes should be grouped in one TB (the optimal value for q)? Second, how does pSHS cope with

high communication latency (TTU)? Again, we first use an example to explain our analysis method. After that, general analytical results are presented.

We use the same example mentioned in last section, in which a 6-word Montgomery multiplication 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.

1) *First step: Assume that $TTU = 0$.* 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 3(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 *allowance time (AT)*. We find that all ATs are no smaller than 0. Under the assumption that $TTU = 0$, 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_{pSHS}) is $24 * CTU$.

2) *Second step: Assume that $0 < TTU < CTU$ and only one transfer exists between two cores.* In this case, all cores

TABLE I
MEANINGS OF SYMBOLS

w	Word length of a processing core (usually 8, 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 $p * q$;
p	Number of cores used for pSHS in the multicore system;
P_i	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 $N = (n_{s-1}, n_{s-2}, \dots, n_1, n_0)$, two operands $A = (a_{s-1}, a_{s-2}, \dots, a_1, a_0)$, and $B = (b_{s-1}, b_{s-2}, \dots, b_1, b_0)$ with $A, B < N$, and the constraint $n' = -n_0^{-1} \pmod{2^w}$. s can be divided by $p * q$. $N, A, \text{ and } B$ are stored locally for each core.

Ensure: $T = (t_{s-1}, t_{s-2}, \dots, t_1, t_0) = A * B * 2^{-n} \pmod{N}$. $n = w * s$. T is stored locally for each core.

The complete Montgomery Multiplication consists of p copies of the following program. On processor P_i , execute:

```

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

```

receiving message with $AT = 0$ have to wait for the messages. In Figure 3(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_{pSHS} is $24 * CTU + 8 * TTU$, shown in Figure 3(b).

3) *Third step: Assume that $TTU > CTU$ 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 3(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. We define the largest TTU in the first piece as the *communication latency tolerance* (CLT). In other words, CLT indicates the largest communica-

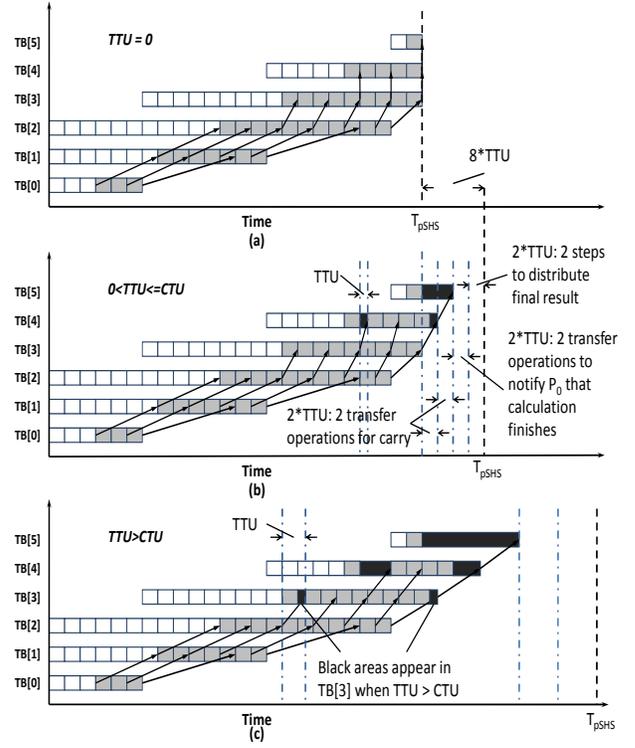


Fig. 3. An example of pSHS in time domain with different TTUs: (a) $TTU = 0$; (b) $0 < TTU \leq CTU$; (c) $TTU > CTU$.

tion latency that guarantees T_{pSHS} 's lowest increasing rate, as TTU increases. For the above example, the communication latency tolerance is CTU . From the following experimental results in Section V, we will see that pSHS has a stable performance when TTU is smaller than CLT.

With the same method, we also performed a general analysis on pSHS. For brevity, we do not include it in this paper. Details can be found in a technical report [14]. The results are as follows.

The CLT of pSHS can be approximated by Equation 1.

$$CLT = \begin{cases} (q - 1) * CTU, & q = 1, 2, 3 \\ q * CTU, & q \geq 4 \end{cases} \quad (1)$$

When TTU is smaller than CLT, the execution time of pSHS can be approximated by Equation 2.

$$T_{pSHS} = \frac{T_{SOS}}{p} + (3p - 1) * TTU + parallel_overhead \quad (2)$$

When TTU is larger than CLT, T_{pSHS} increases more quickly.

Through the above analysis, we have derived the following answers. **Conclusion 1:** We should always choose the largest number for q : (s/p) . The first reason is that the largest q leads to the largest *communication latency tolerance*. The second reason is that a smaller q requires more iterations of the i -loop in Algorithm 2, and therefore the *parallel_overhead* is larger. **Conclusion 2:** The relationship between T_{pSHS} and TTU is shown by Equation 2, given TTU is smaller than the *communication latency tolerance*. When TTU is larger than the tolerance, time cost of pSHS increases more quickly.

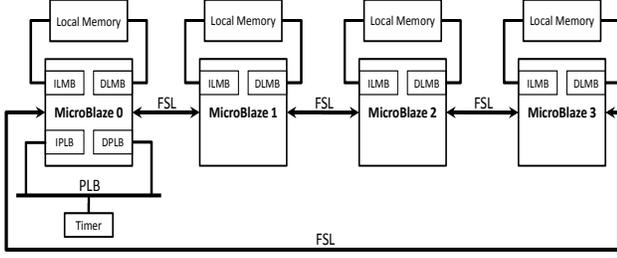


Fig. 4. 4-core architecture with distributed local memory and FSL based message passing connection.

V. EXPERIMENTAL RESULTS

A. Experimental Setup

Experiments are built on a Xilinx Virtex 5 FPGA, where we implemented three parallel platforms 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 4. 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 latency programmable to emulate different communication latencies. 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 of pSHS.

B. Experimental Results

In each platform, we implemented 512 ($s = 16$), 1024 ($s = 32$), and 2048 ($s = 64$) bit-long Montgomery multiplications. We programmed each of these in C according to Algorithm 2. The compiler's optimization level is 2. In every case we tested, $q = s/p$ always leads to the best performance. We show the result with $s = 32$, $p = 4$ in Figure 5.

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 5. Therefore, *our Conclusion 1 that q should be s/p has been demonstrated.* In all the experiments mentioned below, we choose $q = s/p$.

To determine the speedup of the parallel version over the sequential version, we implemented a sequential reference of SOS in MicroBlaze 0. The execution time of that design is listed in Table II.

Since one Montgomery multiplication has $2 * s * s$ boxes, according to Table II, CTU is approximately 26 cycles. We calculate the speedup according to T_{SOS}/T_{pSHS} . The performance of pSHS is shown in Table III.

In Figure 6, the results of Table III are visualized for the case of $p = 4$. Each curve begins with a smaller slope followed by a larger slope after TTU becomes larger than the delay tolerances. Given $q = s/p$, we obtain q equals to 4, 8, and 16

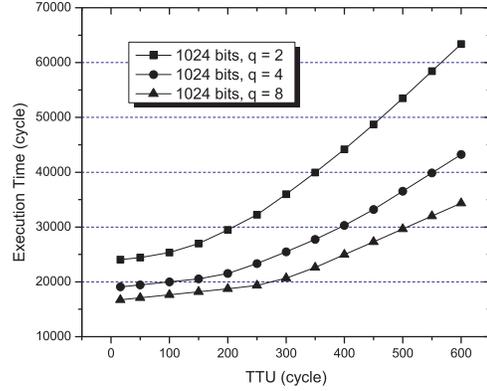


Fig. 5. pSHS's execution time of processing 1024-bit Montgomery multiplication with 4 cores ($s = 32, p = 4$): larger q has better performance and larger communication delay tolerance. Execution time increases faster for smaller q as TTU increases.

for 512-, 1024-, and 2048-bit pSHS respectively. According to the analysis, we get the following *communication latency tolerances* ($q * CTU$): 104, 208, and 416 cycles. In Figure 6, the actual *communication latency tolerances* are 150, 250, and 450 cycles (indicated by arrows). *The results from analysis and experiments are very close, especially for longer operands.* The difference mainly comes from approximations in the analysis process. With TTU smaller than the *communication delay tolerance*, we calculate the slope of each of the curves. The results are all 11, which equals to $3p - 1$ ($3 * 4 - 1 = 11$). Therefore, *Conclusion 2 in Section 4 is demonstrated to be*

TABLE II
EXECUTION TIME OF SOS

operand	512 bits	1024 bits	2048 bits
execution time (cycles)	13953	53905	212145

TABLE III
PSHS'S EXECUTION TIME (ET) & SPEEDUP (SP)

	TTU								
		32	100	300	500	700	900	1100	
$p=2$	512	et	8804	9144	10144	14563	19563	24563	29563
		sp	1.58	1.52	1.38	0.96	0.71	0.57	0.47
	1024	et	29382	29856	30356	31856	34174	42248	50448
		sp	1.83	1.81	1.78	1.75	1.72	1.69	1.67
	2048	et	107228	107568	108568	109568	110568	111568	112568
		sp	1.98	1.97	1.96	1.94	1.92	1.90	1.88
$p=4$	512	et	5680	6054	7154	10472	13972	17472	20972
		sp	2.46	2.30	2.11	1.95	1.60	1.33	1.14
	1024	et	16716	17090	18190	19334	22609	27309	32009
		sp	3.22	3.15	3.06	2.96	2.88	2.79	2.61
	2048	et	56777	57151	58251	59351	60451	62109	66409
		sp	3.74	3.71	3.68	3.64	3.61	3.57	3.54
$p=8$	512	et	4364	4752	6333	9003	11753	14503	17253
		sp	3.20	2.94	2.62	2.20	1.83	1.55	1.34
	1024	et	10762	11150	12300	13450	15486	19036	22586
		sp	5.01	4.83	4.60	4.38	4.19	4.01	3.82
	2048	et	32470	32858	34008	35158	36308	37458	40428
		sp	6.53	6.46	6.35	6.24	6.13	6.03	5.94

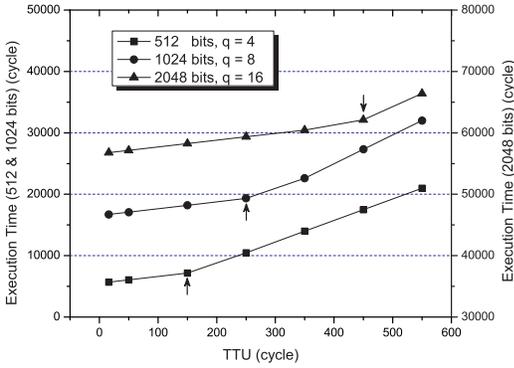


Fig. 6. pSHS's execution time of processing 512-, 1024-, and 2048-bit Montgomery multiplications with 4 cores ($p = 4, q = s/p$).

true for the 4-core architecture. With the same method, we find that it is also true for 2- and 8-core architectures.

Based on 4 cores, the 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.

C. Analysis Based on Results

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 even several hundreds of cycles to transfer one message between neighboring cores is not a difficult requirement for current on-chip multicore 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 [15] [16], pSHS can still be a good candidate for acceleration.

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.

VI. 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.

VII. ACKNOWLEDGEMENT

This research was supported in part through NSF Grant no. 0644070.

REFERENCES

- [1] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobbs's Journal*, vol. 30, 2005.
- [2] K. Sakiyama, L. Batina, B. Preneel, , and I. Verbauwhede, "Multicore Curve-Based Cryptoprocessor with Reconfigurable Modular Arithmetic Logic Units over $GF(2^n)$," *IEEE Trans. on Computers*, vol. 56, no. 9, pp. 1269–1282, 2007.
- [3] J. Fan, K. Sakiyama, and I. Verbauwhede, "Montgomery Modular Multiplication Algorithm for Multi-Core Systems," *IEEE Workshop on Signal Processing Systems*, pp. 261–266, 2007.
- [4] P. Gastaldo, G. Parodi, and R. Zunino, "Enhanced Montgomery Multiplication on DSP Architecture for Embedded Public-Key Cryptosystems," *EURASIP Journal on Embedded Systems*, vol. 8, no. 3, 2008.
- [5] D. Page and N. P. Smart, "Parallel cryptographic arithmetic using a redundant Montgomery representation," *IEEE Trans. on Computers*, vol. 53, no. 11, pp. 1474–1482, 2004.
- [6] R. Szerwinski and T. Güneysu, "Exploiting the Power of GPUs for Asymmetric Cryptography," *Workshop on Cryptographic Hardware and Embedded System (CHES2008)*, LNCS, vol. 5154, pp. 79–99, 2008.
- [7] A. Moss, D. Page, and N. P. Smart, "Toward Acceleration of RSA Using 3D Graphics Hardware," *Cryptography and Coding 2007*, LNCS, vol. 4487, pp. 213–220, 2007.
- [8] S. Fleissner, "GPU-Accelerated Montgomery Exponentiation," *International Conference on Computational Science (ICCS2007)*, LNCS, vol. 4487, pp. 213–220, 2007.
- [9] P. L. Montgomery, "Modular Multiplication without Trial Division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, 1985.
- [10] R. Rivest, A. Shamir, and L. Adleman, "A Method for obtaining Digital Signatures and Public Key Cryptosystems," *Communications of the ACM*, vol. 21, pp. 120–126, 1978.
- [11] National Institute of Standards and Technology (NIST), "Digital Signature Standard (FIPS 186-2)," 2000.
- [12] N. Koblitz, "Elliptic Curve Cryptosystems," *Mathematics of Computation*, vol. 48, no. 177, pp. 203–209, 1987.
- [13] Ç. K. Koç, T. Acar, and B. S. Kaliski Jr., "Analyzing and Comparing Montgomery Multiplication Algorithms," *IEEE Micro*, vol. 16, no. 3, pp. 26–33, 1996.
- [14] Z. Chen and P. Schaumont, "Analysis on pSHS," available: <http://rijndael.ece.vt.edu/chenzm/TRpSHS.pdf>, 2009.
- [15] N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz, "Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs," *Cryptographic Hardware and Embedded Systems (CHES2004)*, LNCS, vol. 3156, pp. 119–132, 2004.
- [16] M. Koschuch, J. Lechner, A. Weitzer, J. Großschädl, A. Szekely, S. Tillich, and J. Wolkerstorfer, "Hardware/Software Co-design of Elliptic Curve Cryptography on an 8051 Microcontroller," *Cryptographic Hardware and Embedded Systems (CHES2006)*, LNCS, vol. 4249, pp. 430–444, 2006.