Multi-exponentiations on Multi-cores

Cem Topçuoğlu
Computer Science and Engineering,
Sabancı University, Istanbul, Turkey
cemtopcuoglu@sabanciuniv.edu

Kamer Kaya
Computer Science and Engineering,
Sabancı University, Istanbul, Turkey
kaya@sabanciuniv.edu

Erkay Savaş
Computer Science and Engineering,
Sabancı University, Istanbul, Turkey
erkays@sabanciuniv.edu

***Abstract*—Modular exponentiation lies at the core of many cryptographic schemes and its efficient implementation is a must for a reasonable practical performance. For various applications, multiple exponentiations with different bases and exponents need to be performed and multiplied. Although this *multi-exponentiation* operation can be implemented by individually exponentiating the bases to their corresponding exponents, as discussed in the literature, a significant performance boost can be obtained when the operation is considered as a whole. However, performing separate exponentiations is pleasingly parallelizable but the latter approach requires a careful implementation on a multi-core processor. In this work, we propose a parallel algorithm and implementation based on an existing multi-exponentiation algorithm with pre-computation. The experimental results show that the proposed implementation is significantly faster than the existing parallel multi-exponentiation schemes in the literature.**

***Index Terms*—Multi-exponentiation, parallel algorithms, multi-core processors.**

I. Introduction

A modular multi-exponentiation takes n bases $X[.]$, n non-negative exponents $Y[.]$, and a modulus m and computes

$$r = \prod_{i=0}^{n-1} X[i]^{Y[i]} \bmod m.$$

Multi-exponentiations frequently arise especially in the cryptography domain; for instance, discrete log-based signature verification schemes requiring a multi-exponentiation with $n = 2$ are very common, e.g. [1]. The batch verification of multiple signatures also requires multi-exponentiation for arbitrary n values [2].

The application we are interested in is *computationally private information retrieval* for which the data is encrypted and outsourced to a server that is responsible to process encrypted queries. The main motivation is keeping the data and access patterns secret to achieve data, query, and response confidentiality. Lipmaa proposed BddCPIR [3] which is a combination of a rooted binary tree and the homomorphic Damgård-Jurik [4] cryptosystem. The key property of Damgård-Jurik is that it provides block-length adjustment capability for the encryptions in different tree levels. In BddCPIR, the leaf nodes hold the data items and the outgoing edges of the internal nodes are labeled as 0 or 1. Thanks to the homomorphic property of Damgård-Jurik, these labels can be used to query an item in an encrypted form.

Recently, instead of the binary trees, we proposed to use octal and hexadecimal trees to reduce the depth of the tree and show that these trees yield a significant improvement on the performasnce [5], [6]. Given the encrypted query bits (encrypted edge labels for each level), starting from the leafs, the server performs a multi-exponentiation for each internal node in a bottom-up fashion where n is equal to the number of children which is 8 or 16 for our case. A multi-core server can concurrently perform independent multi-exponentiations for different internal nodes; for instance, if we have 65,536 data items (leaves), with octal trees there will be $\frac{65,536}{8} = 8,192$ multi-exponentiations for the second level from the bottom and each can be executed with a single thread. However for the root node there is only one multi-exponentiation. Unfortunately, due to the nature of Damgård-Jurik scheme, the complexity significantly increases as the computations proceed from the leaf level to the root level. Thus, this last multi-exponentiation, which is the most expensive one, constitutes a significant response time bottleneck if performed sequentially. Furthermore, for the second level from the top, there will be only eight multi-exponentiations, which are also much more expensive than the ones on the lower levels. Having around 100 cores on a server today, we need to perform each single multi-exponentiation on this level in parallel for faster query processing.

In this work, we propose a parallel multi-exponentiation algorithm with pre-computation and load-balancing mechanism. Although, our application uses $n = 8, 16$, the proposed techniques can be employed for arbitrary large n and for different applications.

The rest of the paper is organized as follows: Section II introduces the notation that will be used in the paper and Section III describes of our parallel algorithm in detail. An improvement on this algorithm for a better load balancing is given in Section IV. Section V provides an experimental evaluation of the proposed techniques and Section VI describes the related work in the literature. Section VII concludes the paper and discusses possible future studies.

II. Notation and Background

In its simplest form, a modular exponentiation takes three integers x, y, and m as inputs, where x is the base, y is the b-bit exponent and m is the modulus, and outputs $r = x^y \bmod m$. One of the earliest algorithms for modular exponentiation traverses the exponent starting from the leftmost (the most significant) bit $y[0]$ to the rightmost bit $y[b-1]$, performs a modular squaring for each bit location and a modular

multiplication for each bit $y[j] = 1$, $0 \leq j \leq b-1$. The pseudocode of this algorithm, EXP, is given in Algorithm 1.

Algorithm 1 EXP(x, y, m, B)

Output: $r = x^y \bmod m$.
1: $r \leftarrow 1$
2: **for** $\ell = 0$ **to** $b-1$ **do**
3: $\quad r \leftarrow (r \times r) \bmod m$
4: $\quad$ **if** $y[\ell] = 1$ **then**
5: $\quad\quad (r \leftarrow r \times x) \bmod m$

The modular multi-exponentiation problem has multiple, n, bases and exponents, and similar to above, a modulus m. Let $X[.]$ be the array storing these bases and $Y[.]$ be the array for exponents. We will use $X[i]$ and $Y[i]$ to denote the ith base and exponent where $0 \leq i < n$. The output of a modular multi-exponentiation function is $r = \prod_{i=0}^{n-1} X[i]^{Y[i]} \bmod m$.

The multi-exponentiation problem can be solved by performing EXP for each ($X[i]$, $Y[i]$, m) triplet and multiplying the results to obtain r. For simplicity, we assume that a modular squaring is implemented via a modular multiplication. Assuming the exponents' bits are equal to 0 or 1 with 1/2 probability, $1.5bn$ multiplications are required for multi-exponentiation with this approach. There exist faster algorithms in the literature to perform a modular multi-exponentiation with less multiplications by processing multiple exponent bits at once. Our work on depends on Lim and Lee's algorithm which employs two phases; pre-computation and multi-exponentiation [7].

Assume that the bases and exponents are partitioned into k groups each having n/k of them. Let x_0, x_1, $\ldots x_{n/k-1}$ be the bases in an arbitrary group. For the ith group, the pre-computation step computes $2^{n/k}$ values

$$P[i][j] = x_0^{e_0} x_1^{e_1} \ldots {x_{n/k-1}}^{e_{n/k-1}} \bmod m$$

where each $e_\ell \in \{0,1\}$, $0 \leq \ell < n/k$, is a binary variable and j is the decimal value of $(e_0 e_1 \ldots e_{n/k-1})_2$. A multi-exponentiation algorithm is given in Algorithm 2, MULEXP, whose pre-computation phase stores each computed value in the $k \times 2^{n/k}$ array P (lines 1-6).

As shown in the pseudocode, each $P[i][j]$ can be computed via a single modular multiplication (lines 4-6). Let $j = (e_0 e_1 \ldots e_{\ell-1} 100 \ldots 0)_2$ be an n/k-bit integer such that e_ℓ is the last 1 in j's binary representation. The algorithm first computes $j' = (e_1 e_2 \ldots e_{\ell-1} 000 \ldots 0)_2$ with the same binary representation as j except that $e_\ell = 0$ (line 4). Then $P[i][j]$ is computed by multiplying $P[i][j']$ with the appropriate base $X[d]$ where $d = \frac{in}{k} + ((n-1) - \ell)$.

After the pre-computation phase, the algorithm performs the multi-exponentiation phase. Similar to EXP, the algorithm MULEXP traverses the exponents from left-to-right but it does not handle each bit individually; instead, it processes a single bit from all group exponents at once. As in Algorithm 1, r is the result which is squared for each bit location $0 \leq \ell < b$ (line 9).

Algorithm 2 MULEXP(X, Y, m, b, n, k)

Output: $r = \prod_{i=0}^{n-1} X[i]^{Y[i]} \bmod m$.
{**Pre-computation phase**}
1: **for** $i = 0$ **to** $k-1$ **do**
2: $\quad P[i][0] \leftarrow 1$
3: $\quad$ **for** $j = 1$ **to** $2^{n/k} - 1$ **do**
4: $\quad\quad j' = (j-1)$ **&** j
5: $\quad\quad d = \frac{in}{k} + ((n-1) - \log(j - j'))$
6: $\quad\quad P[i][j] \leftarrow (P[i][j'] \times X[d]) \bmod m$
{**Multi-exponentiation phase**}
7: $r \leftarrow 1$
8: **for** $\ell = 0$ **to** $b-1$ **do**
9: $\quad r \leftarrow (r \times r) \bmod m$
10: $\quad$ **for** $i = 0$ **to** $k-1$ **do**
11: $\quad\quad j = (Y[\frac{in}{k}][\ell] \ldots Y[\frac{(i+1)n}{k} - 1][\ell])_2$
12: $\quad\quad$ **if** $j > 0$ **then**
13: $\quad\quad\quad r \leftarrow (r \times P[i][j]) \bmod m$

Let $Y[i][j]$ be the jth bit of the ith exponent. Each group i, $0 \leq i < k$ has the exponents

$$Y\left[\frac{in}{k}\right] \text{ to } Y\left[\frac{(i+1)n}{k} - 1\right].$$

For the ℓth location, first $j = (Y[\frac{in}{k}][\ell] \ldots Y[\frac{(i+1)n}{k} - 1][\ell])_2$, which is the decimal value of the binary string obtained by using the ℓth bits of the group exponents, is computed. Then r is multiplied by $P[i][j]$ where this operation accounts for all the modular multiplications due to the ℓth bits of the group exponents in the basic algorithm.

For the pre-computation phase, $k2^{n/k}$ multiplications are required. For the multi-exponentiation phase, assuming the bits are random, $(1 - \frac{1}{2^{n/k}})bk$ multiplications and b squarings are required[1]. Since a squaring is assumed to be implemented via a multiplication, there are

$$k2^{n/k} + \left(1 - \frac{1}{2^{n/k}}\right) bk + b$$

multiplications in total.

To understand the benefits of MULEXP over EXP and the pre-computation, let us consider the case for $n = 16$ and $b = 4096$. With this parameters, the basic algorithm using EXP performs 98304 multiplications. Using MULEXP without a pre-computation phase, i.e., by considering $k = 16$, $n/2 = 8$ multiplications and a single squaring are required per bit which yields 38864 multiplications. With pre-computation, we can process the bits of the exponents at once; with $k = 2$ the proposed scheme performs only 12768 multiplications. The number of groups also affects the performance; for $k = 4$, the number of multiplications is 19520 and for $k = 1$, the pre-computation phase alone requires 65536 multiplications.

III. PARALLELIZATION OF MULTI-EXPONENTIATION ON A MULTICORE SYSTEM

A straightforward parallel solution to the multi-exponentiation problem can be devised via multiple EXP

[1]Assuming the bits are uniformly random are independent, $\Pr(j \neq 0)$ is $\left(1 - \frac{1}{2^{n/k}}\right)$.

executions by assigning each exponentiation to a single thread. The results can then be reduced via $\tau - 1$ modular multiplications where τ is the number of threads. On the other hand, although it is sequentially much more efficient, the parallelization of MULEXP is not that straightforward.

To enable parallelism, we divide the exponent bit locations into τ chunks of equal size as Fig. 1 shows for the first group of bases/exponents. In this scheme, the ith thread processes only its own chunk, compute an incomplete intermediate result r_i and make it complete via extra squarings as shown in the bottom part of the figure. Thanks to these squarings, the partial results can then be combined via modular multiplication. In the example, the bits are divided into four chunks, shown with different colors, where the red-colored bits are highlighted as an example for computing the pre-computed array indices.

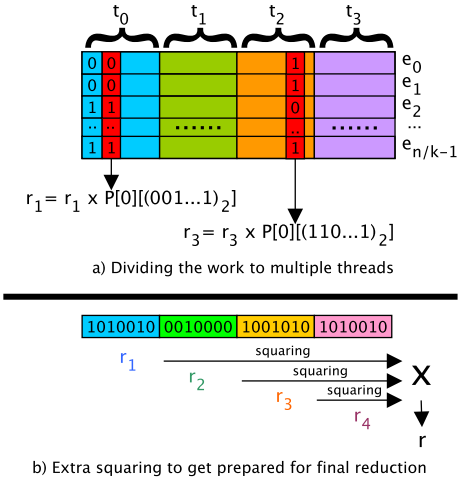


Fig. 1: Parallelization of MULEXP with four threads.

The main problem with the parallelization described above is load balancing; when the chunk sizes are equal, by processing the chunk bits, the incomplete partial results can be obtained with the same number of modular multiplications. However, the later squarings performed to make these results complete make the first thread heavily loaded on the contrary to the last one which does not perform any extra squaring. For instance, with $b = 1024$, two threads, and a single group ($k = 1$), the second thread performs 512 multiplications and 512 squarings, whereas the first one computes 512 multiplications and 1024 squarings. We will solve this problem in the next section by using a scheme that assigns the chunks to the threads in the order of increasing number of bits as shown in Figure 2. With this approach, the threads that require more number of extra squarings will have less exponentiation work hence will start their extra squaring phase earlier. In the next section, we will calculate the exact chunk size for each thread to achieve perfect load balance. But before, let us discuss the parallelization of the pre-computation phase.

A. Parallelization of the pre-computation phase

The pre-computation phase given in Algorithm 2 is optimal in terms of the number of multiplications since this is equal

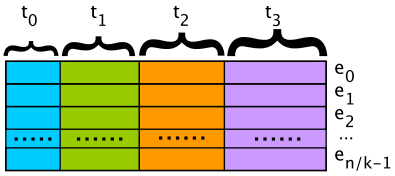


Fig. 2: An imbalanced chunk-to-thread assignment for a load balanced parallel multi-exponentiation scheme.

to the number of values computed. For large k, this phase is easy to parallelize since the pre-computed values for different groups are independent from each other. However, when $k = 1$, which is the case for many applications including ours, an efficient parallel implementation is not straightforward.

For a correct, parallel implementation, the necessary and sufficient requirement is that when a value is being pre-computed for $j = (e_0 e_1 \dots e_{\ell-1} 100 \dots 0)_2$, the value for $j' = (e_1 e_2 \dots e_{\ell-1} 000 \dots 0)_2$ should have been already pre-computed. However, an arbitrary parallelization of the loop at line 1 of Algorithm 2 does not satisfy the requirement since $P[i][j]$ and $P[i][j']$ can be assigned to different threads and their relative computation order cannot be known.

A parallel implementation is straightforward with multiple synchronization points; for instance, if the j values are partitioned with respect to the number of set-bits/ones in their binary representation. With this, starting from the part with the lowest number of set-bits, the parts can then be processed in parallel. However, a synchronization point after each part is necessary. This may create a load imbalance among the threads; for instance, there is only one j with all ones hence, only one task for the corresponding part.

We followed the approach given in Algorithm 3 with no synchronization points. For each group i, a thread first computes $P[i][j]$ from scratch where j has a prefix of $t = \lceil \log(\tau) \rceil$ bits and all zeros after in its base-2 representation. The same thread then computes all the values with the same prefix. In short, the values that will be pre-computed are chunked with respect to their t-bit prefixes and each chunk is assigned to a single thread. This approach presented here is lock-free, it does not require a synchronization mechanism even for a single group of bases/exponents, and it distributes the load to the threads almost evenly (assuming τ, the number of threads is a power of two).

IV. LOAD BALANCED MULTI-EXPONENTIATION WITH PRE-COMPUTATION

Assume that the chunks are distributed as shown in Figure 2. Let t_i be the ith thread for $0 \leq i < \tau$. Let the end bit of the exponent block that will be processed by t_i is equal to $0 \leq \ell_i < b$. For completeness, let $\ell_{-1} = -1$. For the ith thread, the total work in terms of the number of multiplications is

$$\begin{aligned} w_i &= (k+1)(\ell_i - \ell_{i-1}) + ((b-1) - \ell_i) \qquad (1) \\ &= k\ell_i - (k+1)\ell_{i-1} + b - 1 \end{aligned}$$

where in (1), the first part of the sum is for multiplying k pre-computed results with r_i and squaring it. The second part

Algorithm 3 MULEXP(X, Y, m, b, n, k)

Output: $r = \prod_{i=0}^{n-1} X[i]^{Y[i]} \bmod m$.

{**Pre-computation phase**}

1: $t = \lceil \log(\tau) \rceil$
2: **for** $i = 0$ **to** $k-1$ **do**
3: **for** $p = 0$ **to** $2^t - 1$ **in parallel do**
4: Let $(e_0 e_1 \cdots e_t)_2$ be the binary rep. of the prefix p
5: $P[i][j] = \prod_{\ell=1}^{t} X[\ell]^{e_\ell}$
6: **for** $j = 1$ **to** $2^{n/k-t} - 1$ **do**
7: $j' = j + j2^{n/k-t}$
8: $j'' = (j'-1)$ **&** j'
9: $d = \frac{in}{k} + ((n-1) - \log(j' - j''))$
10: $P[i][j'] \leftarrow (P[i][j''] \times X[d]) \bmod m$

{**Multi-exponentiation phase**}

11: $\cdots$ {**same with Algorithm 2**}

is for shifting the exponent via modular squarings and make r_i ready for the reduction step. For a balanced execution, we need to have $w_i = w_{i-1}$ and hence

$$k\ell_i - (k+1)\ell_{i-1} = k\ell_{i-1} - (k+1)\ell_{i-2}$$

which implies

$$\ell_i = \frac{2k+1}{k}\ell_{i-1} - \frac{k+1}{k}\ell_{i-2}. \quad (2)$$

From (2), we have the following theorem:

Theorem 1. *For any number of threads, a balanced execution of modular multi-exponentiation with k groups requires*

$$\ell_i + 1 = \frac{(k+1)^{i+1} - k^{i+1}}{k^i}(\ell_0 + 1)$$

for all $0 \le i < \tau$.

Proof. We will use induction. To show that the statement is correct for $i = 1$, we will use (2):

$$\ell_1 = \frac{2k+1}{k}\ell_0 + \frac{k+1}{k}$$

since $\ell_{-1} = -1$. This implies

$$\begin{aligned} \ell_1 + 1 &= \frac{2k+1}{k}(\ell_0 + 1) \\ &= \frac{(k+1)^2 - k^2}{k}(\ell_0 + 1) \end{aligned}$$

and the base case is shown. As the inductive assumption, let the statement be correct for $i = j$. We will prove that it is also correct for $i = j + 1$. From $w_j = w_0$, i.e., perfect load balance, we have

$$k\ell_{j+1} - (k+1)\ell_j = k\ell_0 + (k+1)$$

and hence

$$\ell_{j+1} + 1 = \frac{k+1}{k}(\ell_j + 1) + (\ell_0 + 1).$$

Using the inductive assumption, we have

$$\begin{aligned} \ell_{j+1} + 1 &= \frac{k+1}{k}\frac{(k+1)^{j+1} - k^{j+1}}{k^j}(\ell_0 + 1) + (\ell_0 + 1) \\ &= \left(\frac{(k+1)^{j+2} - k^{j+2}}{k^{j+1}} - 1\right)(\ell_0 + 1) + (\ell_0 + 1) \\ &= \frac{(k+1)^{j+2} - k^{j+2}}{k^{j+1}}(\ell_0 + 1). \end{aligned}$$

□

Setting $\ell_{\tau-1} = b - 1$ as the last bit location in the exponent, we can derive the value of ℓ_0 as

$$b = \frac{(k+1)^{\tau+1} - k^{\tau+1}}{k^\tau}(\ell_0 + 1)$$

and

$$\ell_0 = \left\lceil b\frac{k^\tau}{(k+1)^{\tau+1} - k^{\tau+1}} - 1 \right\rceil.$$

Similarly, we can compute ℓ_i for $0 \le i < \tau - 1$ as

$$\ell_i = \left\lceil bk^{\tau-i}\frac{(k+1)^{i+1} - k^{i+1}}{(k+1)^{\tau+1} - k^{\tau+1}} - 1 \right\rceil.$$

An additional derivation step reveals that ℓ_i values are non-decreasing as i is increasing since

$$\ell_i = \left\lceil b\frac{k^{\tau-i}(k+1)^{i+1} - k^{\tau+1}}{(k+1)^{\tau+1} - k^{\tau+1}} - 1 \right\rceil,$$

and the only term dependent to i, i.e., the one on the left of the nominator, increases by $(k+1)/k$ fold when i is incremented.

V. EXPERIMENTAL RESULTS

All the simulation experiments in this section are performed on a single machine running on 64 bit CentOS 6.5 equipped with 384GB RAM and a dual-socket Intel Xeon E7-4870 v2 clocked at 2.30 GHz, where each socket has 15 cores (30 in total). Each core has a 32kB L1 and a 256kB L2 cache, and each socket has a 30MB L3 cache. All the codes are compiled with `gcc 4.8.4` with the `-O3` optimization flag enabled. For large number arithmetic, GNU Multiple Precision Arithmetic Library (GMP 6.0), is utilized while OpenMP API is employed for parallel algorithms.

For all the experiments, we performed ten executions and reported the average result. We did not tweak our implementation and set $k = \frac{n}{4}$ to make the size of each group, n/k, always equal to four. With different group sizes, a better performance is possible depending on other parameters.

We start by analyzing the parallel performance of EXP; as expected, when each exponentiation is assigned to a different thread, a linear speedup is obtained as long as the number of exponentiations n is a multiple of the number of threads τ, but this approach can use at most n threads. Table I shows EXP's execution time and perfect scalability with different number of threads. However, as Table II shows, it is promising to consider the multi-exponentiation operation as a whole. The table reports the single-thread execution times for EXP and MULEXP. Although it is sequentially much faster, we know that MULEXP is less scalable than EXP since, with

each additional thread, some amount of extra work is incurred. However, with better load balancing one can keep the approach superior especially when b is large and/or τ is small.

	n	$\tau=1$	$\tau=2$	$\tau=4$	$\tau=8$	$\tau=16$
4096 bits	8	0.31	0.16	0.08	0.04	0.04
	16	0.62	0.31	0.16	0.08	0.04
	32	1.25	0.63	0.31	0.16	0.08
	64	2.50	1.25	0.62	0.31	0.16
8192 bits	8	2.29	1.15	0.57	0.29	0.29
	16	4.63	2.31	1.16	0.58	0.29
	32	9.16	4.58	2.29	1.15	0.57
	64	18.32	9.17	4.58	2.29	1.15

TABLE I: Execution times (in seconds) for EXP with $n \in \{8, 16, 32, 64\}$ bases/exponents and $\tau \in \{1, 2, 4, 8, 16\}$ threads.

		$n=8$	$n=16$	$n=32$	$n=64$
4096 bits	EXP	0.31	0.62	1.25	2.50
	MULEXP	0.12	0.29	0.38	0.71
8192 bits	EXP	2.29	4.63	9.16	18.32
	MULEXP	0.75	1.26	2.27	4.28

TABLE II: Single thread execution times (in seconds) for EXP and MULEXP for $n \in \{8, 16, 32, 64\}$ bases/exponents.

In the rest of this section, we use **v1** to denote the first variant of MULEXP that uses equally-sized chunks for parallelization. The load-balanced variant described in Section IV is denoted as **v2**. Figure 3 shows the normalized execution times of MULEXP variants **v1** and **v2** with respect to EXP, i.e., the runtime of EXP is considered as one second for each case. As the figure shows, thanks to load balancing, the proposed parallel implementation is much better than parallel EXP for all the cases with $b = 8192$. For $b = 4096$, it can be slower when $\tau = 16$. Although larger groups, i.e., smaller k, can work better for some of these cases, we skip this analysis due to the space limitations. The experiments clearly show that load balancing has a significant impact on the performance for all the cases.

To understand the impact of load-balancing better, we measured the number of multiplications, including the squarings, performed by each thread in the multi-exponentiation phase of MULEXP. We report the maximum number of multiplications by a single thread in Figure 4. As the figure shows, the load-balancing mechanism described in Section IV reduces the maximum load, i.e., the number of multiplications, per thread and this is why **v2** takes less time.

To see the relation between the number of multiplications and the performance more clear, we computed time spent per multiplication/squaring by dividing the execution time to the maximum number of multiplications performed by a single thread. Figure 5 shows these times for each experiment setting. This value is computed around 0.011 milliseconds for $b = 4096$ and 0.031 milliseconds for $b = 8192$ and consistent for different n and τ values. Hence, the extra overhead due to extra bitwise operations in our implementation is not significant and the execution time is indeed proportional to the maximum threads load.

To compare the proposed approach with the literature (see next section), we measured the impact of pre-computation by removing it from the implementation of the variant **v2**. Figure 6 shows the execution time of this variant normalized with respect to that of original **v2** with pre-computation. The pre-computation can halve the execution time especially for large b. Again, larger group sizes can yield better execution times.

VI. Related Work

There exist several studies on various variants of the exponentiation problem; a survey on the techniques for fast exponentiation and multi-exponentiation can be found in [8]. These techniques are analyzed and improved on other studies, e.g., [9]. Our work is based on Lim and Lee's multi-exponentiation method which uses pre-computation [7], which we have employed for faster privacy preserving query processing [5], [6] without any parallelization.

Parallelization of the single exponentiation kernel with a smart load balancing mechanism has been studied by Lara et al [10]; we have inspired by this work while developing our algorithms and implementations. Recently, the authors extended their work for the multi-exponentiation kernel [11] which we were aware of at the time of writing. Their implementation does not use pre-computation hence, their load-balancing analyses cannot be applied to our method. Instead, we used a simplified model that exactly computes the amount of work per thread. Furthermore, we also provide a novel parallel pre-computation phase. As the experiments show, the proposed parallel implementation is two times faster than when pre-computation is disabled.

VII. Conclusion and Future Work

In this work, we studied the parallel multi-exponentiation operation on multicore processors. We showed that with pre-computation and a smart load-balancing mechanism, considering the operation as a whole and dividing this task to the cores can be better than the straightforward parallelization with almost linear speedup. One interesting extension is a hybrid form which considers the multi-exponentiation with n bases/exponents as $1 \leq n' \leq n$ tasks for better efficiency; the straightforward approach uses $n' = n$ where the proposed approach uses $n' = 1$. Although the latter form is much faster than the former one on a single core, its efficiency and scalability is much worse. Hence, finding the optimal n' and k values analytically for any n, τ, b combination will be very useful to further reduce the overall execution time. Another extension can be exploiting the techniques such as windowing and deriving the equations for perfect load balancing once these techniques are integrated to the implementation.

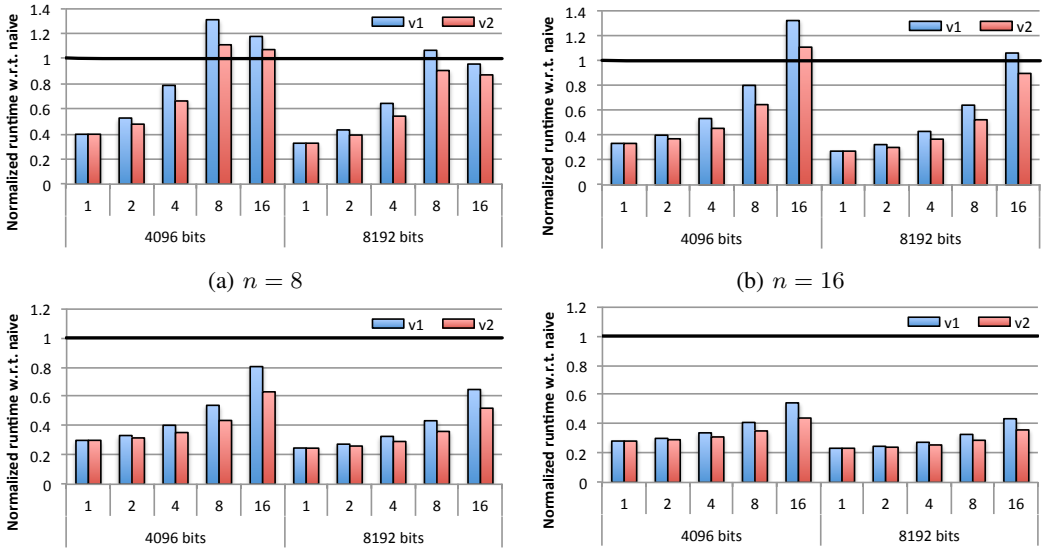
References

[1] D. Naccache, D. M'RaÏhi, S. Vaudenay, and D. Raphaeli, *Can D.S.A. be improved? — Complexity trade-offs with the digital signature standard —*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 77–85.

[2] M. Bellare, J. A. Garay, and T. Rabin, *Fast batch verification for modular exponentiation and digital signatures*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 236–250.

[3] H. Lipmaa, *First CPIR Protocol with Data-Dependent Computation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 193–210.

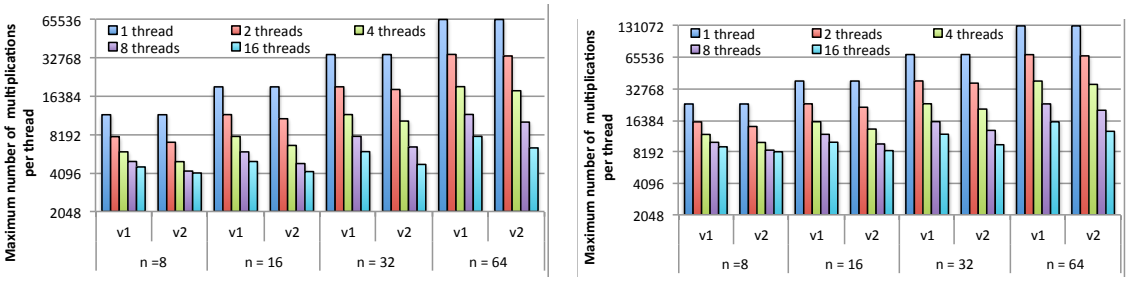
[4] I. Damgård and M. Jurik, "A generalisation, a simplification and some applications of paillier's probabilistic public-key system," in *Proc. of the 4th Int. Workshop on Practice and Theory in Public Key Cryptography*, ser. PKC '01. London, UK, UK: Springer-Verlag, 2001, pp. 119–136.



(a) $n = 8$ (b) $n = 16$

(c) $n = 32$ (d) $n = 64$

Fig. 3: Execution times of MULEXP variants **v1** and **v2** normalized with respect to that of parallel EXP for $b = 4096$ and 8192 and for different values of n; (a) 8, (b) 16, (c) 32 and (d) 64.



(a) $b = 4096$ (b) $b = 8192$

Fig. 4: Maximum number of multiplications per thread with $\tau \in \{1, 2, 4, 8, 16\}$ threads and $n \in \{8, 16, 32, 64\}$ bases/exponents.

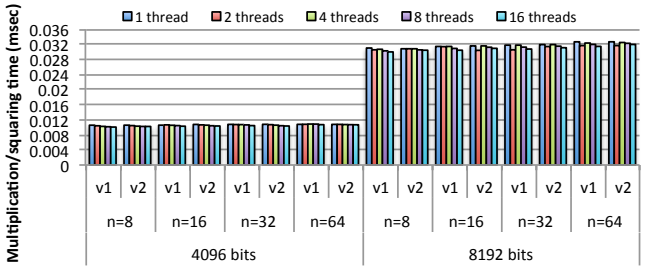


Fig. 5: Time per multiplication for all the experiments: the values are computed by dividing the total execution time to the maximum number of multiplications/squarings perfromed by a single thread.

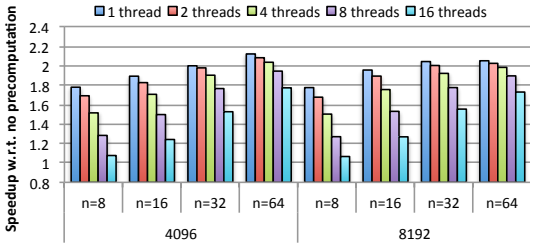


Fig. 6: The execution time of MULEXP-like algorithm without precomputation normalized with respect to that of MULEXP variant **v2**. Both schemes use the balancing approach described in Section IV.

[5] G. Tillem, Ö. M. Candan, E. Savaş, and K. Kaya, *Hiding Access Patterns in Range Queries Using Private Information Retrieval and ORAM*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 253–270.

[6] G. Tillem, E. Savaş, and K. Kaya, "A new method for computational private information retrieval," *The Computer Journal*, no. 13, 2017.

[7] C. H. Lim and P. J. Lee, "More flexible exponentiation with precomputation." in *CRYPTO*, ser. Lecture Notes in Computer Science, Y. Desmedt, Ed., vol. 839. Springer, 1994, pp. 95–107.

[8] D. M. Gordon, "A survey of fast exponentiation methods," *J. Algorithms*, vol. 27, no. 1, pp. 129–146, Apr. 1998.

[9] B. Möller, *Improved Techniques for Fast Exponentiation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 298–312.

[10] P. Lara, F. Borges, R. Portugal, and N. Nedjah, "Parallel modular exponentiation using load balancing without precomputation," *J. Comput. Syst. Sci.*, vol. 78, no. 2, pp. 575–582, Mar. 2012.

[11] F. Borges, P. Lara, and R. Portugal, "Parallel algorithms for modular multi-exponentiation," *Applied Mathematics and Computation*, vol. 292, pp. 406 – 416, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S009630031630474X