Parallel Implementations of S-Vote Electronic Voting Verification and Tallying Processes

Israa A. Saadeh and Gheith A. Abandah

***Abstract*—**Electronic voting systems are being implemented in several countries to provide accuracy and efficiency for the electoral processes with an increased level of security. The Secure National Electronic Voting System (S-Vote) is adopted in this study for its state-of-the-art technologies, privacy, and secure processes. The S-Vote system is a homomorphic e-voting system that uses zero knowledge (ZK) proof protocol to preserve the voter's privacy. Unfortunately, The ZK proofs' is a complex and time consuming protocol which affects the scalability of any homomorphic e-voting system. This study investigates the parallel implementation of the S-Vote verification and tallying processes to reduce the time of vote verification checks especially the ZK proofs verification. Basically, the vote verification process consists of ZK proof, digital signature, and voter eligibility checks. I implement parallelism using java multithreaded programs for parallel program execution. It proposes three parallel implementation schemes for the vote verification and tallying processes which are task, master/slave, and data. The task parallelism spawns a separate thread to perform one of the verification process checks (tasks). The master/slave scheme spawns a thread for each voting kiosk package (client) that performs all the checks. The data parallelism scheme spawns a number of threads equal to the number of physical cores of the tallying machine. Each thread performs the whole verification process checks where the voting kiosk packages are dynamically distributed among them. The obtained results show that the data parallelism scheme is the best. It has the highest relative speedup and efficiency with lowest processing cost. It can verify and tally 64,000 ballots in about 44 minutes with 27.5 relative speedup and 86% efficiency while using 32 threads running on the multi-core tallying machine with 32 cores. The data parallelism scheme reduces ZK proof time. It has a linear speedup with respect to the number of cores and can be used to extend the use of S-Vote system for large electoral processes. For example, using a tallying machine with 128 cores can reduce the verification and tallying processes time for a country as big as Jordan from 25.4 days to 5.7 hours.

***Keywords*—**electronic voting, homomorphic encryption, parallel programming and multithreading, zero-knowledge proof.

This work was supported by the University of Jordan, Faculty of Engineering and technology.

I. A. Saadeh is with the Housing Bank for Trade and Finance and this work is the master thesis research in Network and Computer Engineering Master Program at the University of Jordan (phone: +962790163680; e-mail: srsaadeh@yahoo.com).

G. A. Abandah is professor in the Computer Engineering Department, University of Jordan, Jordan-Amman, (phone: +962798159900; e-mail: abandah@ju.edu.jo).

I. Introduction

Many democratic societies suffer from election fraud, suppression, falsification, and mock elections [1]. They have serious problems throughout their election processes including voter lists manipulation, ballots stuffing, voter intimidation, and vote buying. On other hand, voting centers are often heavily staffed to administer identity check, voting eligibility, and ballot dispersal.

Some staff members unfaithfully enforce regulations for the benefit of their favorite candidates. Identity check is intricate business in cultures where women or men cover their faces. Moreover, primitive techniques are often used to disallow multiple voting such as cutting edge of ID card or dipping voter's finger in special ink.

These societies look forward to new fair election systems that can overcome the traditional election systems weaknesses, prevent electoral fraud, and improve voter participation and trust. The electronic voting systems can become a popular alternative if they satisfy they satisfy the following main challenges [2].

1. Accuracy: count only the valid votes without being tampered with and exclude any invalid vote from the final tally.
2. Democracy: allow only eligible voters to vote and every voter to vote only once.
3. Privacy: do not reveal any voter's choice or allow any voter to prove how he voted. This is to avoid voter intimidation and vote selling.
4. Verifiability: allow anyone to check that each vote was cast by an eligible voter and all votes are correctly counted. In case of electoral disputes, provides means for rechecking the results.
5. Security: always satisfy reliability, availability and data integrity requirements. Additionally, satisfy the accuracy, democracy and privacy requirements and prevent inside or outside attackers from undermining these requirements.
6. Flexibility: support various election types such as parliaments, municipalities, student boards, plebiscites, referendums, etc. Support any eligible voter to vote irrespective of his native language, special needs or literacy level. Allow him to vote in any voting center that is convenient to him. One more aspect; is the flexibility of changing the hardware devices when new or better devices are available.

7. Cost effectiveness: use economic software and hardware components that are important for large-scale elections.
8. Scalability: efficiently carry out various sizes of elections that achieve flexibility, provide better return on investment and facilitate mass quantities production.

Electronic voting schemes are based on blind signatures, homomorphic encryption, or mix-net [3]. The most popular schemes are based on homomorphic e-voting [4]. These schemes count votes without decrypting them. Such systems preserve the voter privacy, but have efficiency problem in vote validity check. Vote validity check often uses zero-knowledge (ZK) proof to verify that each encrypted vote contains valid data without revealing the vote itself [4].

Unlike traditional paper based elections, it is impossible to monitor all electronic operations performed on data from ballot casting to tallying. Accordingly, the validity of votes must be proved by the voters and could be publicly verified. The concept of election verifiability that votes have been recorded, tallied, and correctly declared is called end-to-end verifiability [5]. This verification process unfortunately takes too long computation time that limits the application of e-voting, especially in large-scale elections.

An example of homomorphic-based e-voting system is the Secure National Electronic Voting System [6]. S-Vote achieves the e-voting requirements described earlier, but it is not suitable for large electoral processes due to the long time needed in the verification and tallying processes.

The objective of this study is to reduce the time needed in these processes using parallel implementation. We employ multithreading programing techniques to exploit the parallelism in these processes and execute them in acceptable time on parallel computer. The details of this study are in the master thesis of the first author under the same title and is available through https://theses.ju.edu.jo.

Section II reviews the related work, Section III presents the theoretical background, Section IV describes our implementation of S-Vote, Section V suggests three alternative parallel implementations of S-Vote, Section VI presents the results of evaluating these implementations, and Section VII summarizes the conclusions and suggests future work.

II. Related Work

There has been a number of e-voting systems used in different countries with varying success degrees based on mix-nets and homomorphic voting. Mix-net voting employs a mix network to shuffle the encrypted votes before decrypting them so that the votes cannot be traced back to the voters [7]. Homomorphic voting exploit homomorphism of certain encryption algorithms [8]. Homomorphic voting tallying process costs one single decryption operation for each candidate, so it is more efficient.

The homomorphic cryptosystem of Paillier provides efficient public key cryptography. Its additively homomorphic property can be utilized by secure electronic voting systems [9]. Unfortunately, verification is bottleneck of homomorphic e-voting systems for the cost of its long calculation time.

Many researches aimed to overcome this bottleneck. Reference [8] proposes homomorphic e-voting scheme that adjusts vote format and the corresponding validity check mechanism. A smaller number of checks in larger ranges replaces the large number of checks in small ranges.

Reference [10] reduces the cost of computation and communication by one fourth to one half. So it is still not efficient enough for large-scale election processes. An interesting technique called batched bid validity check was designed in [11] to improve efficiency of bid validity check. It is not a new technique; it is an extension of the traditional batch verification techniques. Meanwhile, this technique has three drawbacks: Firstly, it employs different sealing and parameter settings and cannot guarantee whether it can suit the frequently employed Paillier encryption or its distributed version in homomorphic e-voting schemes. Secondly, it supports one-candidate Yes/No election. Thirdly, it is still not efficient enough for large-scale election applications.

Reference [4] proposes two non-interactive ZK vote validity checks called Protocols 1 and 2. Both protocols can guarantee efficient validity of vote with an overwhelmingly large probability. Protocol 1 modifies and extends the batched bid validity check. It greatly improves the efficiency of the computation of the vote validity check when the voter can select only one candidate. Protocol 2 employs the batched ZK proof too, but it does not limit the number of selected candidates in a vote. Moreover, it needs fewer rounds of communication and has efficient computation.

Reference [12] employed honest verifier ZK proof security model such that the privacy depends on a trust assumption that verifiers are honest. They also proposed a scheme to improve the efficiency of homomorphic e-voting system without optimizing the ZK proof itself. This scheme can only handle a small number of voters. The voters' votes must be grouped. The tallying must separately be carried out in every group. After that, all results will be aggregated to get the final electoral results

S-Vote uses Paillier homomorphic cryptography and the non-interactive ZK Protocol 2 described in reference [4]. S-Vote relies on homomorphic cryptography, distributed key generation, ZK proofs, biometrics, smartcards, open source software, and secure computers to securely and efficiently implement the e-voting processes over the various stages of the electoral process [6].

Reference [9], provided constructions of e-voting system using BGN [5] and Paillier homomorphic cryptosystems. The BGN constructions is only practical for small number of voters and small cipher-text size. Messages computing evaluating are performed in long time. He noted that all constructions are easily parallelized. He assumed that the running time can be reduced directly by using more computers.

Reference [13] introduced Civitas mix-net e-voting system based on [14] cryptographic voting scheme1. The Civitas security is not free. Tradeoffs exist between the levels of

security provided by Civitas tabulation, the time required for tabulation and the tabulation monetary cost. it divided the votes into blocks. The blocks were exploited independently to decrease tabulation time by processing blocks in parallel and giving a set of tabulation teller machines for each block. Tabulation time then does not depend on number of voters. Therefore, performance can scale independently of the number of voters.

Parallel processing is becoming more accessible as the processor manufacturers switch to the model where the microprocessor has multiple processing units (multicores) [15]. The number of cores per processor chip doubles every 18-24 months following Moore's law. Clearly, this change of paradigm has had a huge impact on the software development. Parallel computing can increase the application performance by the execution on multiple cores. However, programmers must reprogram serial application to exploit parallelism [16].

There are two main approaches to parallelize a program: auto-parallelization approach where the sequential program is automatically parallelized using a parallel compiler [17], and parallel programming where programmer modify or develop the application to exploit parallelism. Thus, the program needs to recompile with parallel compiler and no manual modifications are required. However, the amount of parallelism reached using this approach is low due to the complexity of the required automatic transformation. In the parallel programming approach, the application is explicitly modified or developed to exploit parallelism. Generally, this approach obtains a higher performance than auto parallelization one but with the cost of more programing efforts.

Reference [18] presented that the performance of a parallel application depends on the number of threads used to run on a multi-core system. He provided guidelines for finding the appropriate number of threads for getting best performance.

Reference [19] provided tips of motivation showing the relationships between the problem and the various approaches to divide it into parts. These parts are intended to be executed simultaneously via threads to solve the problem

III. THEORITCAL BACKGROUND

This section gives a background on Paillier homomorphic cryptography and ZK proofs used in S-Vote and on parallel application development.

A. Paillier Homomorphic Cryptography

S-Vote uses Paillier key pair cryptosystem for encrypting the voting vectors and decrypting the encrypted tallies. This system has homomorphic addition feature useful in preserving the privacy of votes [9]. Principally, it allows finding sum of the votes by multiplying their encrypted votes (as in (1)). Clear sum can then be decrypted from the encrypted sum (as in (2)).

$$K_V^+(m_1 + m_2) = K_V^+(m_1) \times K_V^+(m_2) \quad (1)$$

$$m_1 + m_2 = K_V^-(K_V^+(m_1 + m_2)) \quad (2)$$

For flexibility, S-Vote allows each voter to select up to O options of C candidates. The vote of each Voter i is encoded as a voting vector $(m_{i,1}, m_{i,2}, \ldots, m_{i,C})$ where $m_{i,j} = 0$ or 1 for $j = \{1, 2, \ldots, C\}$. When the voter choses Candidate j, then $m_{i,j} = 1$ otherwise it is 0. The voting vector is encrypted to $(c_{i,1}, c_{i,2}, \ldots, c_{i,c})$ where the homomorphic property allows finding the encrypted tally of Candidate j from N number of votes through

$$K_V^+\left(\sum_{i=1}^{N} m_{i,j}\right) = \prod_{i=1}^{N} K_V^+(m_{i,j}). \quad (3)$$

As a result, we can find the votes casted for Candidate j by just decrypting the encrypted tally $K_V^+\left(\sum_{i=1}^{N} m_{i,j}\right)$ [9].

B. Zero-Knowledge Proof

As S-Vote requires preserving the voter privacy, zero knowledge proofs are necessary to ensure that encrypted voting vectors carry valid votes. For instance, Voter i can cheat by submitting for his favorite Candidate j the vote $c_{i,j} = K_V^+(100)$ instead of $c_{i,j} = K_V^+(1)$. For this reason, the system requires that each voter must submit his ZK proof.

The advantage of ZK proofs is allowing one party called prover to convince another party called verifier that he knows some secret or knowledge about specific object without revealing what is the object itself.

Non-interactive zero knowledge proofs do not require any interaction between the prover and the verifier where a single message is sent from the prover to the verifier [20]. It is possible to dispose the interaction between prover and verifier when they share a common random public reference string. This is enough to perform zero-knowledge proof check without requiring interaction.

The S-Vote system adopts an efficient honest verifier ZK protocol which is the non-interactive version of Protocol 2 [4]. In this protocol, each Voter i proofs the following two criteria:

$$\wedge_{j=1}^{C}\left(K_V^-(c_{i,j}) = 0 \ \vee K_V^-(c_{i,j}) = 1\right), \quad (4)$$

$$\mathrm{KN}\left[\left(\left(\prod_{j=1}^{C} c_{i,j}\right)/G^{O}\right)^{1/N}\right]. \quad (5)$$

Equation (4) is a proof that every vote in the voting vector is either 1 (for) or 0 (against). Equation (5) is a proof of knowledge of N^{th} root and demonstrates that there are exactly O ones in the voting vector where G and N are part of the cryptosystem public key. Protocol 2 goes through below steps proving that encrypted value $c_{i,j}$ is within the set S of {0, 1}.

Vote Casting:

1. Suppose there are n voters and each voter has to choose O parties from the C candidates.
2. Each voter V_i has his voting vector $(m_{i,1}; m_{i,2}; \ldots m_{i,j} \ldots; m_{i,c})$ where $m_{i,j} = 0$ *or* 1 for $j=\{1; 2; \ldots..; C\}$. A rule is followed: $m_{i,j} = 1$ iff the voter V_i chooses the j^{th} candidate.
3. The voting vector is encrypted to $(C_{i,1}; C_{i,2}; \ldots C_{i,j} \ldots; C_{i,c})$ using homomorphic cryptography where $n=pq$ is the RSA Modulus.

ZK Proof:

The prover generates the proving values and challenge for proving that the encrypted vote $C_{i,j}$ is in the set of {1,0} for *j={1; 2; …..; C}* that shall be sent to the verifier. A security parameter *L* is used and is chosen to be 40 according to Fiat-Shamir heuristic [12], [21]. Since Fiat-Shamir is run for *L* = 20 to 40 executions, the probability for an adversary to fool the verifier for all executions of *L* is very small and does not exceed 2^{-L}.

1. The prover randomly selects the following proving values for *j={1; 2; …..; C}* where:

$$t_{j,0} \in \{0,1,\dots,2^L-1\} \quad (6)$$

$$t_{j,1} \in \{0,1,\dots,2^L-1\} \quad (7)$$

$$challenge\ v \in \{0,1,\dots,2^L-1\} \quad (8)$$

$$v_{j,1-m_{i,j}} \in \{0,1,\dots,2^L-1\} \quad (9)$$

$$r \in Z_N^* \quad (10)$$

2. The prover generates another proving value for *j={1; 2; …..; C}* where:

$$v_{j,m_{i,j}} = v - v_{j,1-m_{i,j}} \bmod 2^L$$

3. The prover generates the commitments that shall be sent to the verifier

$$a = r^N \prod_{j=1}^{C} \left(c_{i,j}\, g^{m_{i,j}-1}\right)^{t_{j,1-m_{i,j}} v_{j,1-m_{i,j}}} \bmod N^2 \quad (11)$$

$$u = r \prod_{j=1}^{C} S_{i,j}^{t_{j,m_{i,j}} v_{j,m_{i,j}}} \bmod N^2 \quad (12)$$

Public Verification:

The verifier calculates the commitments, checks the response, and returns true when they are matched in probability of $1-2^{-40}$ [4].

$$u^N = a \prod_{j=1}^{C} C_{i,j}^{t_{j,0} v_{j,0}} \left(\frac{C_{i,j}}{g}\right)^{t_{j,1} v_{j,1}} \bmod N^2 \quad (13)$$

$$v = v_{j,0} + v_{j,1} \bmod 2^L \text{ for } j = \{1; 2; \dots..; C\} \quad (14)$$

C. Parallel Application Development

There are several aspects that must be considered when developing a parallel application. Mainly, designing the parallel algorithm, implementing the design using a parallel programing language, and evaluating and tuning the developed application.

Designing a parallel algorithm usually follows three main steps: decomposition, scheduling, and mapping. *Decomposition* divides the application computations and data into parts that can be concurrently processed on parallel processors. There are some typical decomposition types such as task, data, recursive, and pipelined [22]. *Scheduling* is the assignment of problem partitions to processes or threads. It specifies the order in which the partitions are executed. *Mapping* is the assignment of threads onto physical computing units (cores) and is usually done by the runtime environment, but sometimes can be influenced by the programmer [23].

The parallel design is coded in a parallel programing language. However, the concurrency in parallel programs often Each voting vector is randomly introducing software bugs such as race conditions. Resource and data dependencies are scheduling constrains and may require a specific execution order of parallel tasks. In this case, synchronization and communication must be put in place [22].

Several metrics are used to evaluate a parallel application. The *parallel execution time* t_p is the time between the start of the application on the first processor and the end of execution on all used processors. It should be smaller than the *sequential execution time* t_s on one processor [24]. Generally, smaller parallel execution times are obtained when the workload is equally distributed among the cores (load balancing). In addition, smaller overhead of data exchange, synchronization, and idle times reduces the parallel execution time. Finding appropriate scheduling and mapping leads to good load balance and small overheads.

The parallel *speedup* measures the increase in speed using multiprocessing and is the ratio of the sequential time to the parallel time $S_R = t_s/t_p$. The *efficiency* measures how efficient the parallel implementation is in using the given parallel resources and is defined by the ratio of the speedup to the number of processors $E_R = S_R/P$. The *cost* is proportional to $t_p \times P$. Finally, the *isoefficiency* scalability metric specifies the rate of workload growth required to keep the efficiency fixed as *P* increases [25].

IV. S-Vote Implementation Details

We implemented the voting, verification, and tallying processes of S-Vote including the technologies necessary for its secure implementation [6]. We developed a Java project using standard Java libraries and components from the homomorphic thep encryption project. These components implement Paillier cryptosystem along with its homomorphic operations, key generation, and zero knowledge proofs http://code.google.com/p/thep.

A. Simulating Voting Process

The voting process is responsible for vote casting and kiosk packages' preparation. Fig. 1 shows the developed procedure used to simulate the S-Vote voting process. For experimentation, we created an eligible voters list consisting of two million national IDs (NID). The list is created once and is arranged in a hash table.

1) *Creating Clear Voting Vector*: created by selecting *O* options out of *C* candidates (*O* number of ones) to produce the clear voting vector of Voter i $V_i = (m_{i,1}, m_{i,2}, \dots, m_{i,C})$.

2) *Creating Encrypted Voting Vectors and ZK Proof (M_i)*: For each vote $m_{i,j}$, this process generates encrypted votes $c_{i,j} = K_V^+(m_{i,j})$ using the public key K_V^+ and generates the ZK proof $P_{i,j}$. It generates first the commitments $u_{i,j}$ for proving that $c_{i,j}$ is in the set {0,1}, which must be sent to the verifier. It then uses

Fiat-Shamir technique to generate the challenge $e_{i,j}$ form $u_{i,j}$ for a non-interactive proof [21].

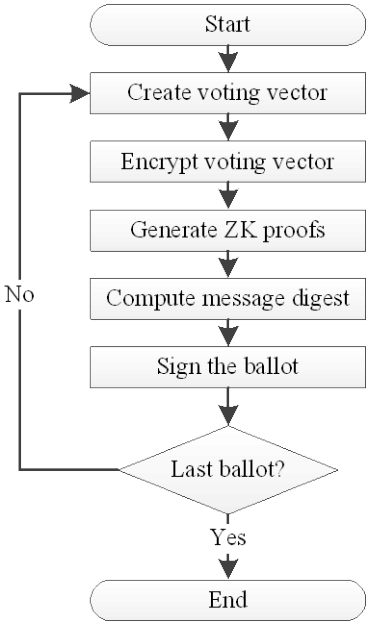

Fig. 1 Simulated S-Vote voting process.

After that, it computes the prover response to $e_{i,j}$ from the random number $r_{i,j}$ then computes $Vs_{i,j}$ and $Es_{i,j}$ values to the verifier that are needed for the last part of the proof. As a result, $P_{i,j}$ is a big integer vector of $[u_{i,j}, e_{i,j}, Vs_{i,j}, Es_{i,j}]$ [6]. Finally, it generates for the voting vector an encrypted value $C_{i,o}$ for the number of options O_i within the Vector V_i and its ZK proof $P_{i,o}$ claiming that the voting vector has O options. Fig. 2 illustrates this process.

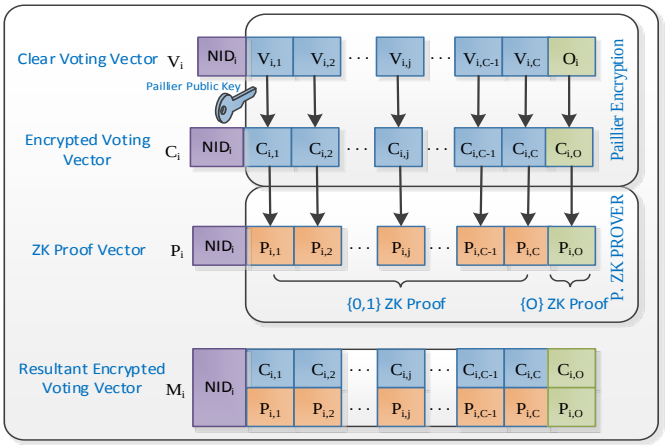

Fig. 2 Encrypting and Adding ZK Proofs for One Voting Vector

3) *Digitally Signing Encrypted Voting Vector:* The voting Message C_i+P_i is digitally signed using SHA-256 hash function H and RSA cryptosystem. Digital signature is required to ensure authenticity and integrity of the encrypted voting vector and its proof. The voter private key K^-_i is kept private while its public key K^+_i is available for verification. The hash function computes message digest D_i then the private key K^-_i signs it. The resulting signature is the *vote receipt* $R_i = K^-_i(D_i) = K^-_i(H(C_i + P_i))$.

After the termination of the voting stage, the kiosk packages are set to the verification and tallying processes. Each kiosk package includes the voter IDs, the encrypted voting vectors C_i, the proofs P_i, and the voting receipts R_i. as in Fig. 3.

B. *Vote Verification Process*

Fig. 4 shows the implemented vote verification and tallying processes. The verification process is responsible for vote verification and starts with preforming the following checks for each voter ballot record.

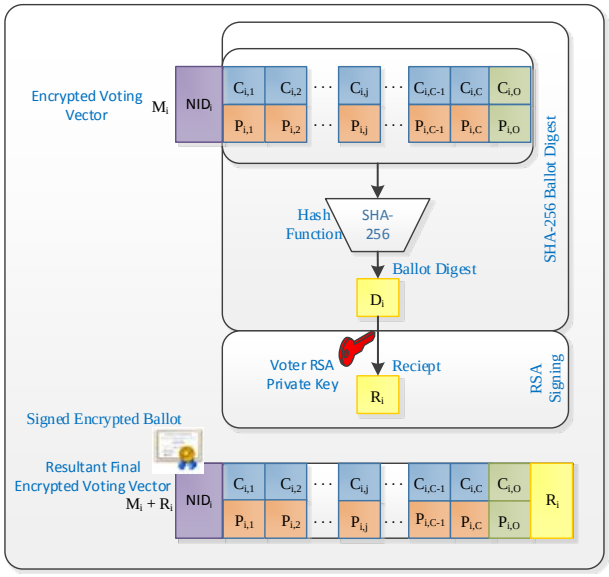

Fig. 3 Adding Digital Signature to Voting Message

1) *Voter Eligibility and Multiple Voting Check:* it is applied on each Voter ballot record. It checks that each voter NID is in the eligible voter list and votes once. Fig.5 shows that how it looks for this NID in the eligible voter NIDs hash table and count the number of its votes. The hash table data structure is used for its good performance.

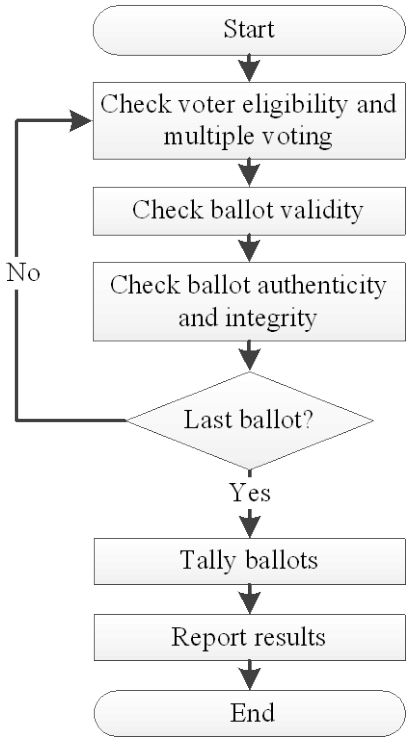

Fig. 4 Implemented S-Vote verification and tallying processes

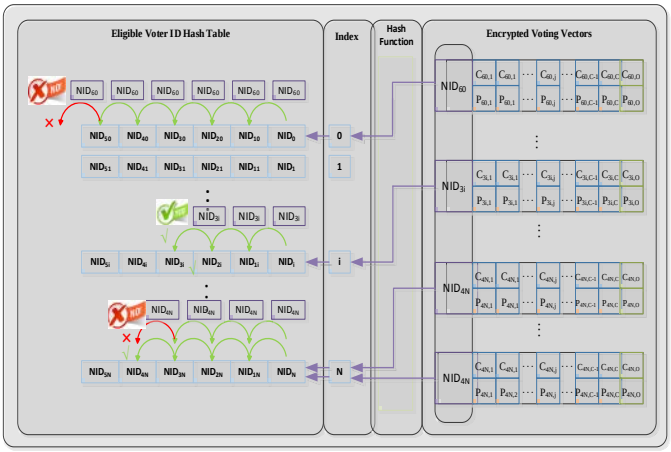

Fig. 5 Voter Eligibility and Multiple Voting Check

1) *Ballot Validity (Zero Knowledge Proof) Check:* It checks the validity of each voting vector such that $c_{i,j}$ is within the set {0, 1} and the number of options in the encrypted voting vector is O. The ZK verifier calculates the proof values $P_{i,j}$ for each $c_{i,j}$ using $u_{i,j}$ generated by the prover then checks the response from the prover $Vs_{i,j}$, $Es_{i,j}$ and $e_{i,j}$ using the Fiat-Shamir heuristic which returns true if it is OK and accept the prover claim that $c_{i,j}$ in the set of {0, 1}, otherwise it returns false. The same check is performed to accept the prover claim that voting vector has O options by $P_{i,o}$ as described in Fig. 6.

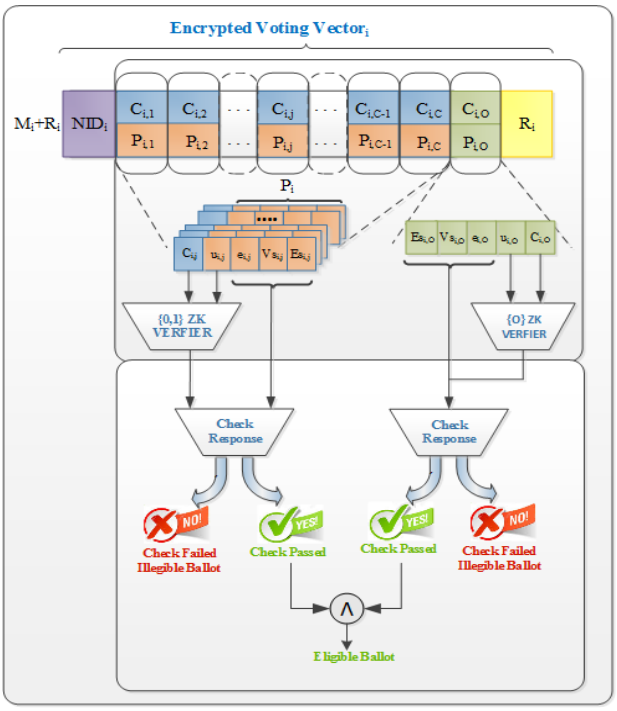

Fig. 6 Ballot Validity Check (ZK Proof)

C. *Tallying Process*

In the tallying process, only encrypted ballots that pass all these checks are considered eligible voting vectors and are passed to the tallying process. Fig. 7 shows the tallying process that consists of:

1) *Ballot Tallying:* It finds the final encrypted tally T_j for each Candidate j by calculating the product of pass votes casted for this candidate $T_j = \prod^{N}_{i=1} c_{i,j}$ (Eq. (3)).

2) *Result Decryption:* It decrypts the final tally T_j for each candidate using the distributed voting private key $R_j = K^{-}_{V}(T_j)$. Finally, the final election results are announced.

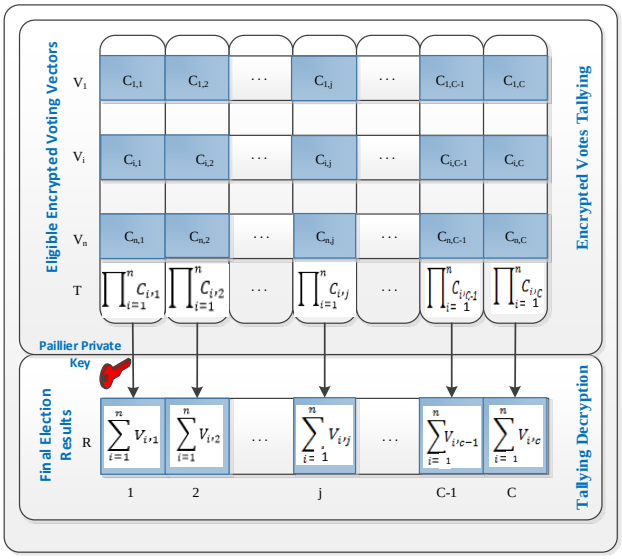

Fig. 7 Tallying Process and Final Tally's Decryption

V. Parallel Implementations of Verification and Tallying Processes

We used Java multithreaded programs to speed up the verification and tallying processes. The following three parallel schemes are implemented and evaluated.

A. Task Parallelism Scheme

The ballot verification process has multiple checks (tasks): voter eligibility, multiple voting, ballot authenticity, and vote validity checks. When you think in parallel execution, you start with dividing the problem into tasks. Accordingly, the vote verification checks are divided into tasks. Each task is responsible for one check and runs through a separate thread with the intention that all threads work on the entire data set, but each thread does a specific task.

This scheme is known as task decomposition. The verification function is divided into four separated sub functions and the kiosk packages (ballots) are given to all threads for processing. The final tallying process starts after the finish of the last running thread.

B. Master/Slave Parallelism Scheme

In the Master/slave scheme, the master class spawns one slave thread to handle each kiosk package. This is some kind of data parallelism where data is partitioned to slaves and each slave handles one kiosk package independently, performs all vote verification checks, and exists when it is done. The number of slave threads equals the number of packages. The final tallying starts at the finish of the last slave.

C. Data Parallelism Scheme

When having many kiosk packages (data) to process, we can divide this large set of data among multiple threads. This concept is knowing as data parallelism in which each thread does the same work but on its subset of data. Supercomputers have excelled at for years. In presence of this, the numbers of simultaneously running threads will be equivalent to the numbers of physical cores to get higher efficiency. Each thread performs the verification and tallying processes as the way as the sequential implementation does but on a sub set of data. The kiosk packages (data) will be dynamically distributed among the different threads in a round robin manner during the run time.

Master class spawn threads as many as the number of physical cores. Each running thread asks for an available kiosk package to process. Accordingly, this class updates the global kiosk counter and gives the requester thread an available package. The kiosk assignment step is synchronously executed so that only one thread can be served at a time to keep data consistency.

Java lock() and unlock() method are used to control the access to this shared resource by the multiple threads. Commonly, lock() grants on thread at a time an exclusive access to kiosk assignment procedure, and it is released for another thread by unlock(). Finally, by the end of the last kiosk package processing, the electoral process ends.

VI. Experimental Result

In this section, we present the evaluation results of the sequential and three parallel schemes. We also evaluate and discuss the best adopted parallel scheme.

All experiments were done on a server with four 8-core Intel Xeon processors E7-8837, 2.67 GHz clock rate, and 256 GB memory. The Java JRE 1.7 project is compiled using GNU Complier for Java (GCJ) version 4.8 and run on an Ubuntu 12.04 virtual machine. The host is Windows Server 2012 R2 Datacenter Edition. The virtual machine monitor is Microsoft Hyper-v 2012 R2.

The selected number of ballots per kiosk package is 500 ballots, the number of candidates is 16, and the number of options is 4. For this configuration, the ballot size is 52 KB: each encrypted vote $c_{i,j}$ along with its ZK proof is 3 KB, the encrypted number of options and its ZK proof is 3 KB, and the digital signature is 1 KB. This size, as a function of the number of candidates, is:

$$\text{Ballot Size (KB)} = 3C + 3 + 1 = 3C + 4. \quad (15)$$

Additionally, Runtime.getMemory() java instrumentations are used to estimate the memory usage for verifying a ballot. 273 KB is occupied to verify a ballot and 308 MB for creating the NID hash table with 2,000,000 entries.

A. Sequential Implementation Results

Fig. 8 shows the execution time of the sequential implementation of the verification and tallying processes as a function of the number of ballots. As expected, there is a direct linear relationship between the execution time and the number of ballots. This implies that we can predict the execution time for any large election. For example, processing 2 million ballots would take 627 hours (15 days).

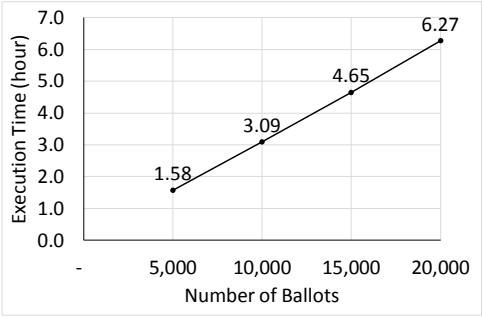

Fig. 8 The execution time of the sequential implementation

B. Alternative Parallel Schemes Results

We present here the results of evaluating the three parallel schemes: Task, Master/slave, and Data. We evaluate them by finding their execution times relative to the sequential one using four cores and number of ballots B = 20,000. Fig. 9 shows the speedup of these three schemes.

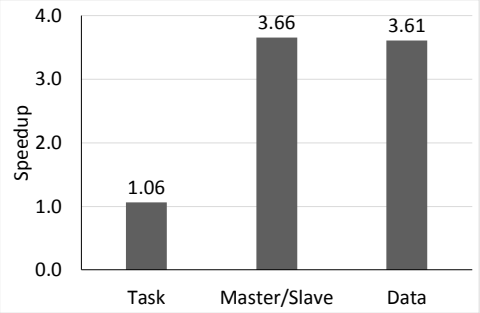

Fig. 9 Parallel schemes speedups with four cores

Task has the worst speedup. This scheme does not divide the problem into equivalent tasks as shown in Fig. 10, which shows the breakdown of the sequential time shown in Fig. 8. The vote validity check involving the long ZK proof computations takes about 95% of the time. Tallying and authenticity and eligibility checks take the rest 5%. Therefore, the thread responsible of the vote validity task has much more load than other threads and finishes long after they finish.

On other hand, Data and Master/server show good speedups, implying high efficiencies and low costs. Although Master/slave shows the best speedup here, it is not a scalable solution. The cost of thread creation increases as the number of kiosk packages increases and the computer will straggle with large number of threads handling many packages concurrently. Excessive number of live threads leads to performance degradation due to competition on limited number of cores, available memory, and other resources. Running this program on a large number of packages crashes the system when it runs out of virtual memory.

The Data scheme is the most preferable scheme. Beside its good performance, it limits the number of spawned threads, and proportionally maps threads to physical cores. It dynamically balances the load as packages are distributed on demand to free threads. Accordingly, we adopt this scheme.

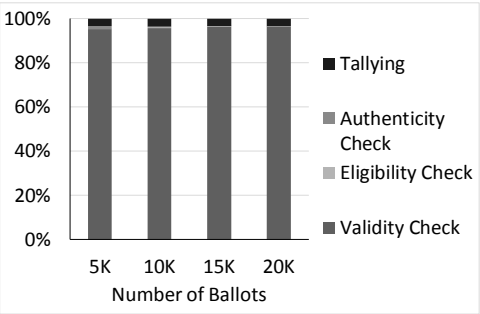

Fig. 10 The breakdown of the sequential execution time

C. Data Parallelism Scheme Evaluation

We present here an evaluation for the Data scheme on a varying number of cores and a number of ballots B = 64,000. The number of spawned threads is selected to match the number of available cores. Table I shows the parallel execution time, speedup, efficiency, and cost of this scheme compared with the sequential implementation. As Fig. 11 illustrates, a sub-linear speedup is achieved as we increase the number of cores. As the speedup and efficiency drop with more cores, the cost increases. The number of threads are increased to match the number of available cores. Thus, the assigned workload becomes smaller as the number of threads increased and reduces the overall computation time. Based on this, we can determine the hardware specifications requirements for serving a number of ballots within 8 hours for example.

Table I Evaluation of data scheme

Cores	Time (hr)	Speedup	Efficiency	Cost
Sequential	19.50	-	-	19.50
4	4.99	3.9	98%	19.95
8	2.67	7.3	91%	21.32
16	1.32	14.8	92%	21.13
32	0.71	27.5	86%	22.68

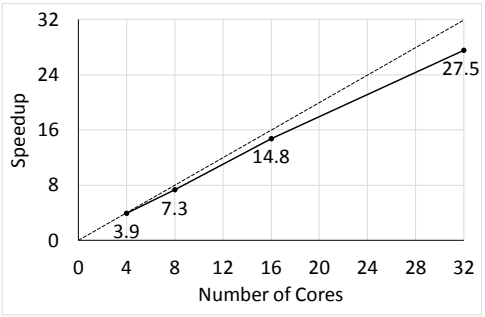

Fig. 11 Data parallelism speedup.

On the other hand, when the number of threads is increased, the synchronization delay becomes larger and the contention on the memory and storage increases. In average, each thread will ask for a kiosk package K/T times. The thread waits up to T time units till it has an exclusive access to kiosk assignment procedure. Thus, each thread has K overhead time. For this reason, Fig. 12 shows a slight degradation in parallelization efficiency as the number of threads increases due to the synchronization of kiosk assignment.

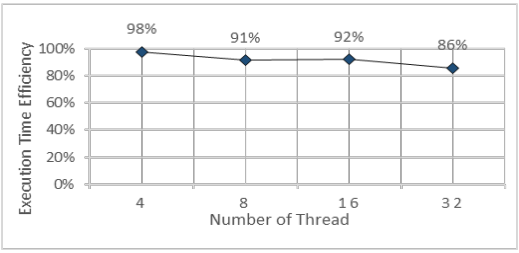

Fig. 12 Data Parallelism Efficiency

The efficiency decreases with more cores because the ballot packages are divided among more cores resulting in smaller fraction of the parallel region to the serial region.

Moreover, a set of experiments are conducted to check the scalability of data parallelism scheme based on the isoefficiency metric. Fig. 13 plots speedup against number of cores for different values of B up to 32 cores.

The result listed in table II and shown in Fig.13 show the efficiency of the 16 configurations and illustrate two things.

First, for a given problem instance, the speedup does not keep the linear increase as the number of cores increases beyond the assigned workload. The speedup curve tends to saturate as in the instance of processing 8,000 ballots on 32 cores. In other words, efficiency drops with increasing the number of cores.

Second, a larger number of *B* yields higher efficiency for the same number cores. Given that increasing the number of cores reduces efficiency and increasing the size of the computation increases efficiency, it should be possible to keep the efficiency constant by increasing both the size of the problem and the number of cores simultaneously to consider that the parallel system is scalable. For instance, Table II shows the efficiency of verifying and tallying 8,000 ballots on tallying machine with four cores is 86%. If the number of cores is increased to eight, the number of ballots is scaled up to verify and tally 16,000 ballots efficiency remains in average of 86%.

Thus, Although the efficiency drops with more cores, it increases with larger numbers of ballots. Note that if the number of ballots is increased proportional to the number of cores (diagonal cases), efficiency stays constant about 86%.

Accordingly, the S-Vote with data parallelism is scalable since the efficiency of data parallel execution maintained at a constant value by simultaneously increasing the number of cores and the number of ballots being verified.

Table II Isoefficiency of the data scheme

Ballots	**4 cores**	**8 cores**	**16 cores**	**32 cores**
8K	**86%**	72%	68%	34%
16K	89%	**87%**	81%	69%
32K	94%	89%	**85%**	77%
64K	98%	91%	92%	**86%**

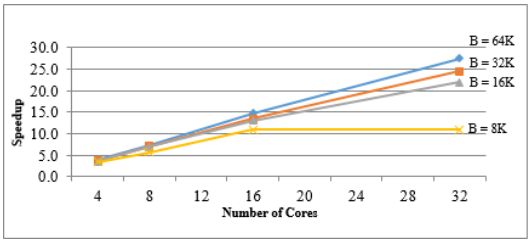

Fig. 13 Data Parallelism Isoefficiency Metric

D. Studying the Effect of Number of Candidates and Options

As another point of view, we study the factors that may affect the ZK proof itself. The next two experiments are performed to study the effects of numbers of the candidates *C* and options *O* on the ZK proof check.

The experiment is running on 32 cores using the data parallelism scheme, *B*=64,000, and *O*=2.

Fig. 14 shows that the ZK proof has a direct linear relationship with the number of candidates. The ZK proof is a function of the number of votes per ballot. This true as the ZK proof is conducted for each encrypted vote within the ballot and its execution time proportioned to the increase number of votes within the ballots.

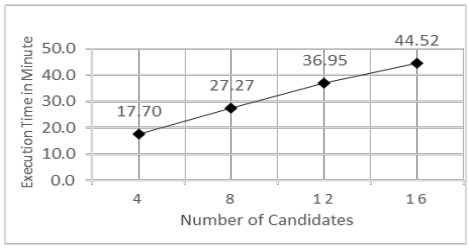

Fig. 14 Number of Candidates Effect on the ZK Proof

On the other hand, the number of options per ballot does not affect the ZK proof as only one proof is required to ensure that there are *O* options in each ballot regardless what is the value of *O* is as demonstrated in Fig. 15. The experiment is running on 32 cores and number of ballots *B*=64,000.

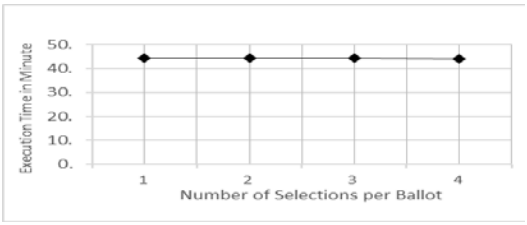

Fig. 15 Number of Options Effect on the ZK Proof

E. Discussion

The execution time is clearly dependent on the number of ballots. The Data scheme can reduce this time to an acceptable time using sufficient number of cores. The size of the parallel computer can be determined according to the election process size and time allowed to process all ballots. In a country like Jordan, we need enough cores to process around 2 million ballots in less than eight hours. Recalling from Table I that 32 cores process 64,000 ballots in 0.71 hour, then using 128 cores allows processing these ballots in 5.5 hours (0.71 × (2,000,00/64,000) × (32/128) = 5.5).

We have also evaluated the effects of changing the number of candidates and the number of options on the execution time. We found that the execution time linearly depends on the number of candidates, but is not affected by the number of options.

VII. CONCLUSION AND FUTURE WORK

S-Vote homomorphic e-voting system is adopted in this study for its e-voting requirements satisfaction. S-Vote uses public key cryptography, hashing techniques, homomorphic cryptography, and zero knowledge proofs for achieving the e-voting requirements of privacy, authentication, and validation of data integrity

We have implemented the voting, verification, and tallying processes of the S-Vote e-voting system. The sequential implementation shows that the verification and tallying processes take long execution time and have linear relationship with the number of voters. We have implemented three parallel schemes aimed to reduce this execution time: Task, Master/slave, and Data. The Task scheme is not scalable and is inefficient due to the load unbalance of the various verification

tasks. Although the Thread scheme shows good speedup with small number of ballot packages, it crashes with large numbers. In Data scheme, a number of threads equals the number of physical cores are spawned and they dynamically request and process ballot packages until all packages are processed. This scheme shows good speedup and efficiency and scales well as the number of ballots and the number of cores are increased. This scheme processes 64,000 ballot using 32 cores in 0.71 hours with 27.5 speedup and 86% efficiency. Therefore, a large national election of 2 million ballots can be processed in an acceptable time of 5.5 hours using 128 cores.

The cost of processing cores is decreasing with time. However, we could get higher speedups and large cost reduction by porting the Data scheme to a graphics processing unit (GPU). Many applications have shown great speedups by exploiting the thousands of processing units now available on high-end GPUs.

For future work, we are looking forward to extend our work to perform full implementation of S-Vote system. Beside of the parallel implementation for vote validity and authenticity checks, we would like to cover all aspects of S-Vote proposed components including the distributed key generation, threshold cryptography, kiosk design, and smartcard implementations.

ACKNOWLEDGMENT

I sincerely thank my supervisor Dr. Gheith Abandah for his guidance and support throughout this research. His advice, comments, discussions, and interpretations were very beneficial for my completion of this study. I learned from his insight a lot. I believe that I learned from the best and I deeply appreciate it.

REFERENCES

[1] M. Allansson, J. Baumann, S. Taub, L. Themnér, and P. Wallensteen, “The first year of the Arab Spring,” SIPRI Yearbook: Armaments, Disarmament and International Security, 2012, pp. 45-56.

[2] T. Antonyan, S. Davtyan, S. Kentros, A. Kiayias, L. Michel, N. Nicolaou, A. Russell, and A. Shvartsman, “State-wide elections, optical scan voting systems and the pursuit of integrity,” IEEE Trans. Information Forensics and Security, vol. 4, no. 4, 2007, pp. 597–610.

[3] A. Huszti, “A homomorphic encryption-based secure electronic voting scheme,” Publ. Math. Debrecen, vol. 79, no. 3-4, 2011, pp. 479-496.

[4] K. Peng and B. Bao, “Efficient vote validity check in homomorphic electronic voting,” Int’l Conf. Information Security and Cryptology, 2008, pp. 202-217.

[5] B. Adida, Advances in Cryptographic Voting Systems, Doctoral Diss., Massachusetts Institute of Technology, Cambridge, 2006.

[6] G. Abandah, K. Darabkh, T. Ammari, and O. Qunsul, “Secure national electronic voting system,” J. Information Science and Engineering, vol. 30, no. 5, 2014, pp. 1339-1364.

[7] C.A. Neff, “Verifiable mixing (shuffling) of ElGamal pairs,” VHTi Tech. Doc., VoteHere, Inc., 2003.

[8] J. Groth, “Non-interactive zero-knowledge arguments for voting,” Int’l Conf. Applied Cryptography and Network Security, pp. 467–482, 2005.

[9] P. Paillier, “Public key cryptosystem based on composite degree residuosity classes,” Int’l Conf. Theory and Applications of Cryptographic Techniques,1999, pp. 223-238.

[10] K. Chidaand G.Yamamoto,"Batch processing for proofs of partial knowledge and its applications," IEICE Trans. Fundamentals E91CA, 2008, pp.150–159.

[11] K.Peng,C.Boyd,, and EDawson, "Batch verification of validity of bids in homomorphic e-auction," Springer Heidelberg, 2007, pp. 209–224.

[12] K. Peng and F. Bao, “Efficient proof of validity of votes in homomorphic e-voting,” 4th Int’l Conf. Network and System Security,2010, pp. 17-23.

[13] M. Clarkson, S. Chong, and A. Myers, "Civitas: Toward a secure voting system. Security and Privacy, Security and Privacy," IEEE, Oakland, 2008, pp. 354 – 368

[14] D. Catalano, A. Juels, and M. Jakobsson, "Coercion-resistant electronic elections," Computer and Communications Security, New York, USA, 2005, pp.61–70.

[15] D. Kirk and D. Hwu, Programming Massively Parallel Processors: A Hands-On Approach, Morgan Kaufmann, San Francisco, 2010.

[16] W. Hwu, K Keutzer, and T. Mattson, “The concurrency challenge,” IEEE Design and Test of Computers, vol. 25, no. 4, 2008, pp. 312-320.

[17] H. Kasim, V. March, R. Zhang, and S. See, “Survey on parallel programming model,” IFIP Int’l Conf. Network and Parallel Computing, 2008, pp. 266-275.

[18] K. Pusukuri, R. Gupta, and L. Bhuyan, "Thread reinforcer: Dynamically determining number of threads via OS level monitoring Workload Characterization (IISWC)," IEEE International Symposium, 2011, pp. 116 –125.

[19] J. Diaz, C. Munoz-Caro, and A. Nino, "A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era," IEEE Trans. Parallel and Distributed Systems, vol. 23, 2012, pp. 1369-1386,S

[20] A. De Santis, G. Di Crescenzo, R. Ostrovsky, G. Persiano, and A. Sahai, “Robust non-interactive zero-knowledge”, Annual Int’l Cryptology Conf., 2001, pp. 566-598.

[21] A. Fiat and A. Shamir, “How to prove yourself: Practical solutions to identication and signature problems,” Conf. Theory and Application of Cryptographic Techniques, 1986, pp. 186-194.

[22] D. Jacobsen, J. Thibaulty, and I. Senocak, “An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU cluster,” 48th AIAA Aerospace Sciences Meeting and Exhibit, 2010.

[23] T. Rauber and G. Runger, Parallel Programming for Multicore and Cluster Systems, Springer Science & Business Media, Berlin, 2010.

[24] B. Barney, Introduction to parallel computing, Lawrence Livermore National Laboratory, https://computing.llnl.gov/tutorials/parallel comp, 2012.

[25] A. Grama, A. Gupta, and V. Kumar, “Isoefficiency: Measuring the scalability of parallel algorithms and architectures,” IEEE Parallel & Distrib. Technol, vol. 1, 1993, pp. 12-21.