TELKOMNIKA, Vol. 15, No. 4, December 2017, pp. 1884 ∼ 1893
ISSN: 1693-6930, accredited A by DIKTI, Decree No: 58/DIKTI/Kep/2013
DOI: 10.12928/telkomnika.v15.i4.6049

A Load-Balanced Parallelization of AKS Algorithm

Ardhi Wiratama Baskara Yudha and Reza Pulungan[‡]
Department of Computer Science and Electronics, Faculty of Mathematics and Natural Sciences,
Universitas Gadjah Mada, Yogyakarta, Indonesia
[‡]Corresponding author, e-mail: pulungan@ugm.ac.id

Abstract

The best known deterministic polynomial-time algorithm for primality testing right now is due to Agrawal, Kayal, and Saxena. This algorithm has a time complexity $\mathcal{O}(\log^{15/2}(n))$. Although this algorithm is polynomial, its reliance on the congruence of large polynomials results in enormous computational requirement. In this paper, we propose a parallelization technique for this algorithm based on message-passing parallelism together with four workload-distribution strategies. We perform a series of experiments on an implementation of this algorithm in a high-performance computing system consisting of 15 nodes, each with 4 CPU cores. The experiments indicate that our proposed parallelization technique introduces a significant speedup on existing implementations. Furthermore, the dynamic workload-distribution strategy performs better than the others. Overall, the experiments show that the parallelization obtains up to 36 times speedup.

Keyword: *Primality testing, AKS algorithm, parallelization, load balancing, high-performance computing.*

Copyright © 2013 Universitas Ahmad Dahlan. All rights reserved.

1. Introduction

Prime numbers are the cornerstone of number theory. Mathematicians and number theorist, since ancient times, have been fascinated by many problems concerning prime numbers. In modern time, many of the most important cryptographic algorithms rely on big prime numbers to perform encryption and decryption. One of them is Rivest-Shamir-Adleman (RSA) algorithm [1], which is now widely used in storage encryption [2], digital certificate [3], and web security [4], including in banking transaction security. RSA algorithm depends on the fact that it is difficult to find the prime factors of a big integer. Electronic Frontier Foundation (EFF) offers $250,000 as a reward to the first individual or group who discovers a prime number with at least 1,000,000,000 decimal digits [5]. Searching for a prime number is usually based on an efficient algorithm that determines whether a given number is prime or composite. Such algorithms are called *primality testing* algorithms.

Most of primality testing algorithms are probabilistic, namely they cannot ascertain the primality of a given number, but only provide a probability that the given number is prime. Miller-Rabin primality test [6, 7], for instance, has an error rate below 25%, which means that if the given number passes this test n times, then the probability that the number is prime is $1 - 0.25^n$ [8]. Solovay-Strassen [9] primality test, on the other hand, has an error rate below 50%. Probabilistic primality testing algorithms are relatively fast, of low complexity, but with tunable accuracy. However, there are cases that require certainty that a given number is prime or not; and thus, probabilistic algorithms cannot be used.

In 2002, three Indian computer scientists Agrawal, Kayal, and Saxena [10] proposed a deterministic—*i.e.*, non-probabilistic—primality testing algorithm that runs in polynomial time; we will refer to this algorithm as *AKS algorithm*. This is the first deterministic polynomial-time algorithm for primality testing. Since this seminal paper, the primality testing problem no longer resides in the complexity classes of NP-Hard, NP, or ZPP [11]. AKS algorithm, interestingly, is relatively simple and straightforward, while previous work by other researchers attempted to show that primality testing is of polynomial time complexity by making complex modifications on existing primality testing algorithms [12].

Received March 20, 2017; Revised October 26, 2017; Accepted November 14, 2017

Since this theoretical breakthrough, many researchers have proposed theoretical and practical improvements to this algorithm soon after it was released in public. Notable among them are Lenstra [13] and Bernstein [14]. Bernstein [14] proposed two practical possibilities for accelerating AKS algorithm with low-level speedup by improving the integer squaring method and high-level speedup by reducing the number of for loop iterations in the last step of the algorithm. This included all state-of-the-art improvements on reducing the last for loop iterations and produced speedup of many orders of magnitudes. These have been incorporated in the latest version of AKS algorithm.

Lenstra and Pomerance [15, 16], on the other hand, proposed theoretical improvements to AKS algorithm and obtained a new technique with time complexity $\mathcal{O}(\log^6(n))$. They modified the original AKS algorithm by decreasing the number of iterations in the for loop. This is done by replacing the use of the cyclotomic polynomials in AKS algorithm by a monic polynomial $f(x)$ of degree r with integer coefficients such that the ring $\mathbb{Z}[x]/(f(x), n)$ is a pseudofield. Bernstein in [17] proposed a further theoretical improvement to AKS algorithm with time complexity $\mathcal{O}(\log^4(n))$. The proposal also attempted to decrease the number of iterations in the for loop by replacing the use of the cyclotomic polynomials by random Kummer extensions of $\mathbb{Z}[x]/n$.

Crandall and Papadopoulos [18] implemented a variant of AKS algorithm by Lenstra [13] and found that empirically the time complexity of the variant is $c\log^6(n)$, where c is around 1,000 clock cycles. Li [19] also implemented the Lenstra variant of AKS algorithm using C++ and NTL library to handle the polynomial data structure. In this implementation, a 15-decimal-digit prime number required around 3,000 seconds to compute in a single-processor computer. Menon in [20] implemented AKS algorithm in SAGE (Software for Algebra and Geometry Experimentation), and produced an implementation, in which a 25-decimal-digit prime number required more than 4,000 seconds to compute in a single-processor computer. Cao [21] analyzed the storage space requirement for AKS algorithm and showed that the required storage space for testing a number with length 1,024 bits is about 1,000,000,000 Gigabyte, which is practically infeasible. This is due to the need to store extremely large polynomials during the computation.

Many scientists have made improvements on the original AKS algorithm, but sequential implementations of the algorithm are still impractical to use due to the expensive computation involved in each step and its storage requirements. Future direction seems to be towards parallel implementations. This paper reports on our effort to develop a parallelization technique for AKS algorithm based on message-passing parallelism (using MPI) and to find out the best workload-distribution strategy for the parallelization technique.

The rest of the paper is organized as follows: Section 2 presents the basis of AKS algorithm. Section 3 describes the proposed parallelization technique, together with four accompanying workload-distribution strategies. In Section 4, we present the result of our experiments with the proposed parallelization technique and the four workload-distribution strategies and provide analysis. Section 5 concludes the paper.

2. Preliminaries

Let $\mathbb{Z}$ be the set of integers and let a and b be two positive integers. Let $gcd(a, b)$ be the greatest common divisor of a and b. The two integers a and b are relatively prime if and only if $gcd(a, b) = 1$. Let $\phi(a)$ be the Euler's totient function, namely the number of positive integers smaller than a that are relatively prime to a. For relatively prime a and r, let $o_r(a)$ be the order of a modulo r, namely the smallest integer k such that $a^k \equiv 1 \pmod r$. Let $a \text{ rem } b$ be the remainder of integer division between a and b.

In the earliest version of their publication, Agrawal, Kayal, and Saxena obtained an algorithm with the worst-case time complexity of $\mathcal{O}(\log^{12}(n))$, where n is the given number. In this paper, we are referring to the latest version (version 6) of their publication [10], in which the latest AKS algorithm was presented. The latest version has incorporated many improvements proposed by many researchers and the resulting algorithm runs in polynomial time with the worst-case complexity of $\mathcal{O}(\log^{15/2}(n))$. Prior to the publication of this algorithm, there were other primality proving algorithms that seemed to run in polynomial time, but AKS algorithm is the first one that is de-

terministic as well as of polynomial time [18]. The main idea of AKS algorithm is described in Lemma 1, which is a generalization of Fermat's little theorem.

Lemma 1 ([10]) *Let $a \in \mathbb{Z}$ be relatively prime to $n \in \mathbb{Z}$ and $n \geq 2$. Then n is prime if and only if:*

$$(x + a)^n \equiv x^n + a \pmod{n}. \tag{1}$$

To reduce the number of operations performed, both sides of (1) can be simplified by taking their respective remainders modulo a polynomial $x^r - 1$, for some small positive $r \in \mathbb{Z}$, namely:

$$(x + a)^n \equiv x^n + a \pmod{x^r - 1, n}. \tag{2}$$

However, right now the bi-implication in Lemma 1 no longer applies, since non-prime n may also satisfy (2) for some a and r. Theorem 1—as reformulated by Granville in [12]—forms the cornerstone of AKS algorithm. The theorem basically asserts that for appropriately selected r's, if (2) is satisfied by some a, then n must be a prime. Therefore, r must be selected accordingly.

Theorem 1 ([10, 12]) *Given $n \in \mathbb{Z}$ and $n \geq 2$, let $r < n$ be a positive integer satisfying $o_r(n) > \log^2(n)$. Then n is prime if and only if:*

(1) n is not a perfect power,

(2) n does not have any prime factor $\leq r$, and

(3) $(x + a)^n \equiv x^n + a \pmod{x^r - 1, n}$ for any integer a, where $1 \leq a \leq \sqrt{\phi(r)} \log(n)$.

A straightforward implementation of Theorem 1 is given in Algorithm 1, where condition (1) corresponds to the first if; and conditions (2) and (3) correspond to the first and the last for, respectively.

Algorithm 1: AKS algorithm

Input: $n \in \mathbb{Z}, n \geq 2$
Output: A string Prime or Composite

```
1  begin
2    if n = a^b, where a, b ∈ Z and a, b > 1 then
3      return Composite
4    end
5    Find the smallest r that satisfies o_r(n) > ⌊log^2(n)⌋
6    for 2 to r do
7      if gcd(a, n) > 1 then
8        return Composite
9      end
10   end
11   for a ⟵ 1 to ⌊sqrt(φ(r)) log(n)⌋ do
12     if (x + a)^n ≢ x^n + a (mod x^r − 1, n) then
13       return Composite
14     end
15   end
16   return Prime
17 end
```

3. Proposed Method

3.1. Parallel AKS Algorithm

Scrutinizing Algorithm 1, we can see that the algorithm basically comprises 4 steps: determining whether n is a perfect power (lines 2–4); determining r (line 5); determining whether n has prime factors $\leq r$ (lines 6–10); and determining the congruence of polynomials $(x + a)^n$ and $x^n + a$ modulo $(x^r - 1, n)$ (lines 11–15) for some values of a. Of these four steps, the last takes most of the computation times of the algorithm, since we are dealing with an enormous n. Furthermore, when raising polynomial $(x+a)$ to the power n—albeit modulo $(x^r - 1, n)$—intermediate results might be enormous polynomials requiring large storage and heavy computation. Our parallelization effort will be focused on computing this last step. Parallelizing the other steps will incur communication overhead that, with the current state of networking technology, renders the saving achieved by the parallelization worthless even for hundreds-decimal-digit n.

As has been noticed by Crandall and Papadopoulos in [18], AKS algorithm is an embarrassingly parallel algorithm. It can easily be parallelized using master-slave technique, by distributing the work of determining the congruence of polynomials $(x + a)^n$ and $x^n + a$ modulo $(x^r - 1, n)$ for different values of a to different computer nodes in a message-passing parallel system. Figure 1 illustrates this master-slave technique.

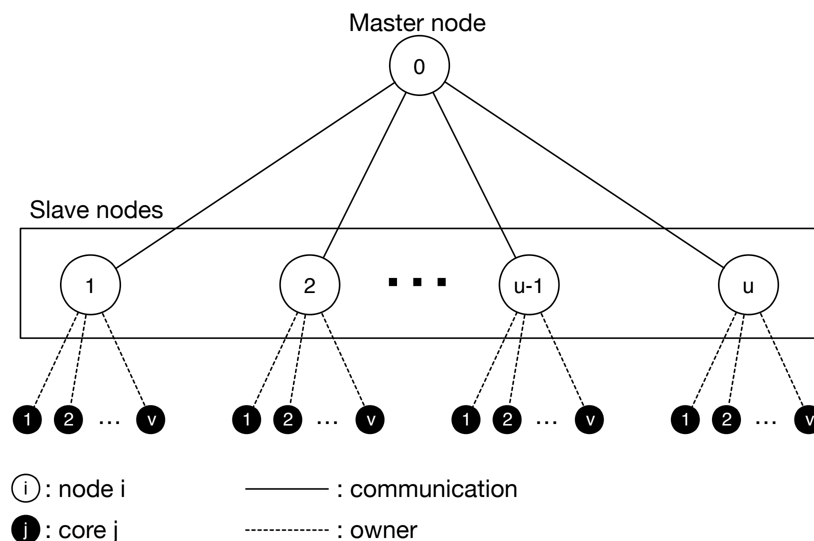

Figure 1. The design of the parallelization technique

In the beginning, the master node performs the computation of the first three steps of AKS algorithm sequentially. Once the master node obtains the value of r, it broadcasts the values of n and r together with other necessary information about distribution of work (namely the distribution of the values of a) to all slave nodes. Each slave node then proceeds with the computation of determining the congruence of polynomials $(x + a)^n$ and $x^n + a$ modulo $(x^r - 1, n)$ for several values of a.

A slave node communicates only with the master node and only in two cases: (1) when for some value of a the polynomials are not congruent, and (2) when for all values of a assigned by the master node, both polynomials are always congruent, and thus signaling that the work assigned to the slave node has been completed. Upon receiving a communication of type (1) from a slave node, the master node immediately dismisses the last for loop and thereby announces that n is composite; and proceeds to command the rest of the slave nodes to abort their computation. Receiving communication type (2) from all slave nodes indicates that all slave nodes have completed their work and all of them find that the two polynomials are congruent for all values of a; the master node then proceeds to announce that n is prime.

A modern computer system usually has multi-core CPUs. A parallelization technique where each of these CPU cores in a slave computer node is treated as a slave node as well is referred to as *single-level parallelization*. In this technique, each core is assigned by the master node several values of a to compute separately from other cores in the same computer node. Communications from all cores in a slave node to the master node must pass through the same channel of communication and this may result in contention. However, compared to the computation time spent for each value of a, the overhead produced by this contention is negligible.

3.2. Workload-Distribution Strategies

The single-level parallelization technique requires the distribution of workload from the master node to all slave nodes. This basically entails distributing the values of a for slave nodes to work on. Recall from Algorithm 1 that the congruence of polynomials $(x + a)^n$ and $x^n + a$ modulo $x^r - 1$ must be determined for $1 \leq a \leq \lfloor \sqrt{\phi(r)} \log(n) \rfloor$. Let $q = \lfloor \sqrt{\phi(r)} \log(n) \rfloor$ and u be the number of slave nodes. Further, let $\%$ stand for the integer division operator. In the following, we present four workload-distribution strategies that will be experimented on in this study.

Strategy 1 Figure 2 illustrates the first workload-distribution strategy. A rectangle in the figure represents a single value of a, while the circle right below the rectangle represents the slave node responsible for computing that value. This strategy is the simplest of the three strategies, where slave node i is responsible to determine the congruence of polynomials $(x+a)^n$ and x^n+a modulo $x^r - 1$, for $(i - 1)(q\%u) + 1 \leq a \leq i(q\%u)$. Hence slave node #1 works on the first $q\%u$ values of a, slave node #2 works on the second $q\%u$ values of a, and so on, while slave node #u works on the remaining values of a. This last slave node may only work on fewer than $q\%u$ values of a, if q is not divisible by u.

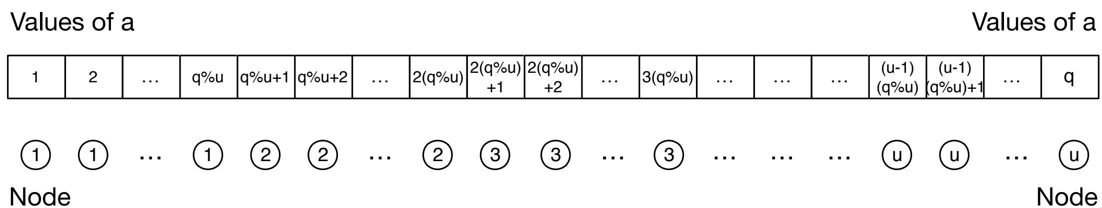

Figure 2. Workload-distribution strategy 1

Strategy 2 One of the main concerns with the first strategy is that one slave node is assigned only with values of a that are consistently smaller or larger than those assigned to other slave nodes. All values of a assigned to slave node #1, for instance, are smaller than those assigned to slave node #2. A larger value of a *may* result in a longer computation time, since the resulting intermediate polynomials will have larger coefficients, which in turn take longer to multiply and require more storage. The second and third strategies try to address this.

Figure 3 illustrates the second workload-distribution strategy. The first slave node will get $a = 1$, the second slave node will get $a = 2$, and so on, until the last slave node will get $a = u$. This is then repeated until all values of a are exhausted. Therefore slave node i will be assigned the values of a of $i, i + u, i + 2u, \ldots, i + ju$, where j is the largest integer that still satisfies $i + ju \leq q$. This strategy manages to avoid assigning one slave node values of a that are consistently smaller or larger than those assigned to other slave nodes. However, each value of a assigned to a slave node is always relatively smaller or larger than that assigned to other slave nodes. For every value i assigned to slave node #1, for instance, the value $i + 1$ is assigned to slave node #2. Hence, if larger value of a *always* results in longer computation time, slave node #1 will complete its workload earlier than slave node #2. This problem will be addressed by the third strategy.

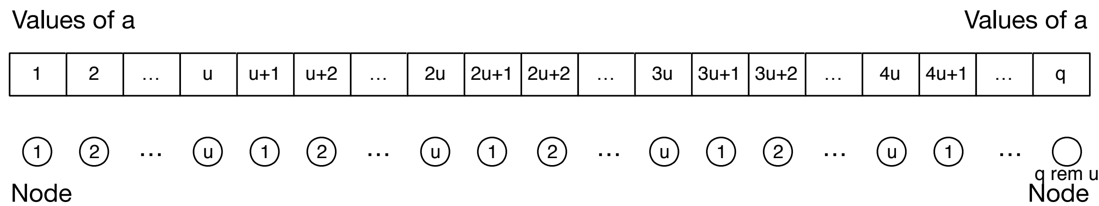

Figure 3. Workload-distribution strategy 2

Strategy 3 The third strategy addresses the problem encountered in the second strategy by ensuring that if a slave node is assigned a small value of a, it will be compensated by another assignment with large value of a. Figure 4 illustrates the third workload-distribution strategy.

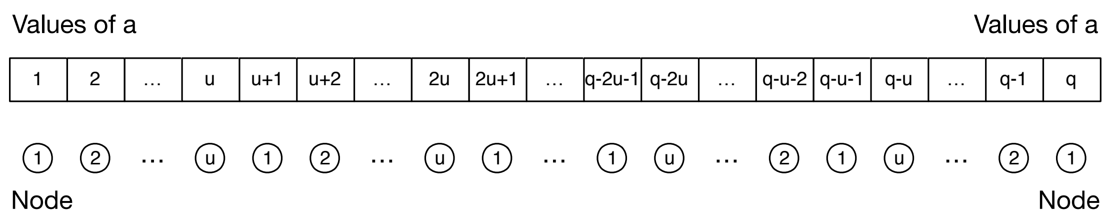

Figure 4. Workload-distribution strategy 3

Hence, since slave node #1 is assigned the value $a = 1$ (the smallest), then it will also be assigned the value $a = q$ (the largest). Similarly, since slave node #2 is assigned the value $a = 2$ (the second smallest), then it will also be assigned the value $a = q - 1$ (the second largest). This is carried out subsequently until all values of a are assigned to all slave nodes in similar fashion: if the value $a = i$ is assigned to slave node j, then the value $a = q - i$ is also assigned to slave node j, for $i \leq q\%2$.

Strategy 4 All previous strategies are static, in the sense that workload distributions are actually predefined even before execution; a specific node always gets the same set of a's when the input n is the same. In this fourth strategy, we propose a dynamic strategy where the set of a's assigned to a slave node cannot be predicted before it is run. The idea is, firstly, a slave node is assigned an a according to its id, for example slave node #1 gets $a = 1$, slave node #2 gets $a = 2$ and so on until slave node #u gets $a = u$. After completing a work, a slave node requests to the master node for another remaining a or informs the master node if the polynomial congruence check produces false result. The master node then sends a remaining a to the requesting node or the master node simply terminates all nodes and output composite result for the other condition. When no remaining a exists, the master node terminates all slave nodes then outputs prime result.

4. Result and Analysis

4.1. Implementation

Since we are primarily concerned with big numbers, we use GNU Multiple Precision (GMP) arithmetic library version 6.10 to handle integer of arbitrary length. In the first step of the algorithm, GMP function $\mathrm{mpz_perfect_power_p()}$ is used to check for perfect powers. To compute the value of r in the second step, we use function $\mathrm{PowerMod()}$ of NTL library version 9.10.0, which basically performs integer modular exponentiations. For checking the existence of factors of the input number that are no more than r in the third step, NTL function $\mathrm{GCD()}$ is used.

Communications between master and slave nodes is performed using MPICH library version 3.2. To broadcast the input number n and the value of r, the master and slave nodes use function $\mathrm{MPI_Bcast}()$. Since the MPI does not support data type $\mathrm{mpz_t}$ defined by GMP as well as data type ZZ defined by NTL, n and r are first converted to arrays of bytes before they are broadcast. Once arrived, they will be converted back to type $\mathrm{mpz_t}$ using function $\mathrm{mpz_init_set_str}()$. After all the values required to compute the last step are obtained by a slave node, it then computes the left and right side of the congruence using function $\mathrm{PowerMod}()$.

4.2. Experimental Setup

All experiments in this study are conducted using High-Performance Computing (HPC) system provided by Directorate of Information System and Resources (DSSDI) of Universitas Gadjah Mada. The HPC system has 15 slave nodes, each with 2 CPU Dual Core AMD Opteron^{TM} Processor 280 (hence, 4 CPU cores), 4 GB DDR3 RAM, OpenSUSE 11.2 64 bits operating system, and GCC compiler version 6.1.

We experiment on prime numbers ranging from 5 digits to 35 digits in length as shown in Table 1. The seven prime numbers selected are the largest prime numbers for the corresponding numbers of digits according to [22].

Table 1. Prime numbers used in experiments

Digits	Prime Number
5	99,929
10	9,999,999,929
15	999,998,727,899,999
20	99,999,999,999,999,999,989
25	9,989,999,899,883,889,989,999,899
30	909,090,909,090,909,090,909,090,909,091
35	68,476,562,763,327,854,359,085,599,065,855,383

4.3. Result

Comparing the workload-distribution strategies Table 2 shows the running times of the sequential as well as the parallel implementations of AKS algorithm for the seven prime numbers. The parallel implementations are run on a 60-processor message-passing system, while the sequential one is run on one of the processors. It is evident that the dynamic workload distribution (strategy 4) performs consistently and significantly better than other strategies in all experiments, which means that this strategy is the most load balanced among the proposed strategies. This also indicates that the overheads associated with communication times between the master node and slave nodes are insignificant compared to the computation times for different values of a.

From Table 2, it is clear that patterns from the running times of the first three workload-distribution strategies are not easy to discern. This result is contrary to the authors' original expectation, as described in Section 3.2. The result indicates that the computation times required for the values of a are not proportional to those values: a larger value of a may require less computation time than that of smaller one.

Speedups for various number of processors The previous result shows that workload distribution strategy 4 produces the best parallel implementation for AKS algorithm. In this part, we focus on this strategy and find out the speedups that are achievable for various number of processors. The result is presented in Figure 5.

Figure 5 shows that for almost all numbers of digits, speedup mostly grows linearly as the number of processors used in the computation increases. The apparent exception to this is for

Table 2. Running times (in seconds) of the different workload-distribution strategies

Digits	Sequential	Parallel			
		Strategy 1	Strategy 2	Strategy 3	Strategy 4
5	1.57168	0.41278	0.19912	0.20151	0.11407
10	105.7	5.3	2.1	2.2	1.7
15	712.2	35.3	22.6	24.0	19.6
20	3,236.0	128.2	130.4	128.8	112.3
25	8,848.8	446.6	371.5	374.1	325.4
30	32,343.2	1,421.6	1,317.3	1,972.4	1,179.6
35	70,901.7	4,121.8	4,177.6	4,152.3	2,457.4

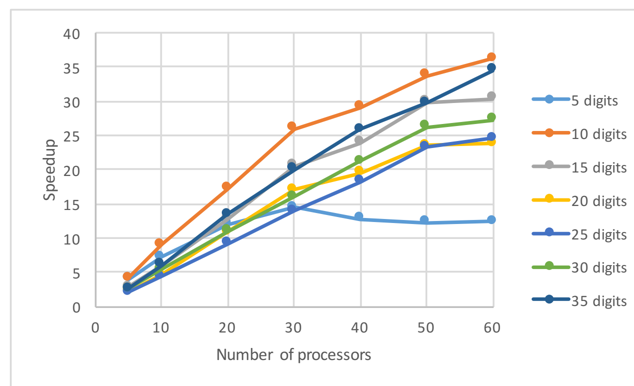

Figure 5. Speedups obtained by varying the number of processors

when the number of digits is 5. For this case, there are only 275 different values of a to check and each of them requires a relatively short computation time. When the number of processors exceeds 30, communication overheads becomes large enough to offset the savings of computation times by the parallelism. Overall, the largest speedup is obtained when the number of digits is 10 and it is a bit more than 36 times the computation time of the sequential implementation.

The effects of multi-core processors The high-performance computing system used in the experiments consists of computers, each with multi-core processors. When each of this core is treated as a node, contentions may occur when there is more than one core communicating simultaneously with the master node. In this part, for each workload-distribution strategy, we vary the number of cores per node used in the parallel computation to establish their effects on the overall computation time. For this purpose, three scenarios are created, namely 1 core per node, 2 cores per node, and 4 cores per node. In all of these scenarios, the overall number of cores is maintained at 8 in order to set a baseline. Figure 6 depicts the result of the experiments.

From Figure 6, we can conclude that workload distribution strategies 1, 2 and 3 are almost not affected by the number of cores per node used in the parallel computation. This is understandable since, in these strategies, a slave node rarely communicates with the master node. Communications between a slave node and the master node occurs only during termination, namely when the slave node finds that the polynomials are not congruent for a specific value of a or when it finds that the polynomials are congruent for all assigned values of a.

The effect for workload-distribution strategy 4, however, is stark and the larger the prime number the more pronounced the effect. Having more cores per node results in longer computation time. This is in line with our expectation, since, having more cores per node results in heavier use of the communication line between the master and the slave nodes. What we do not expect

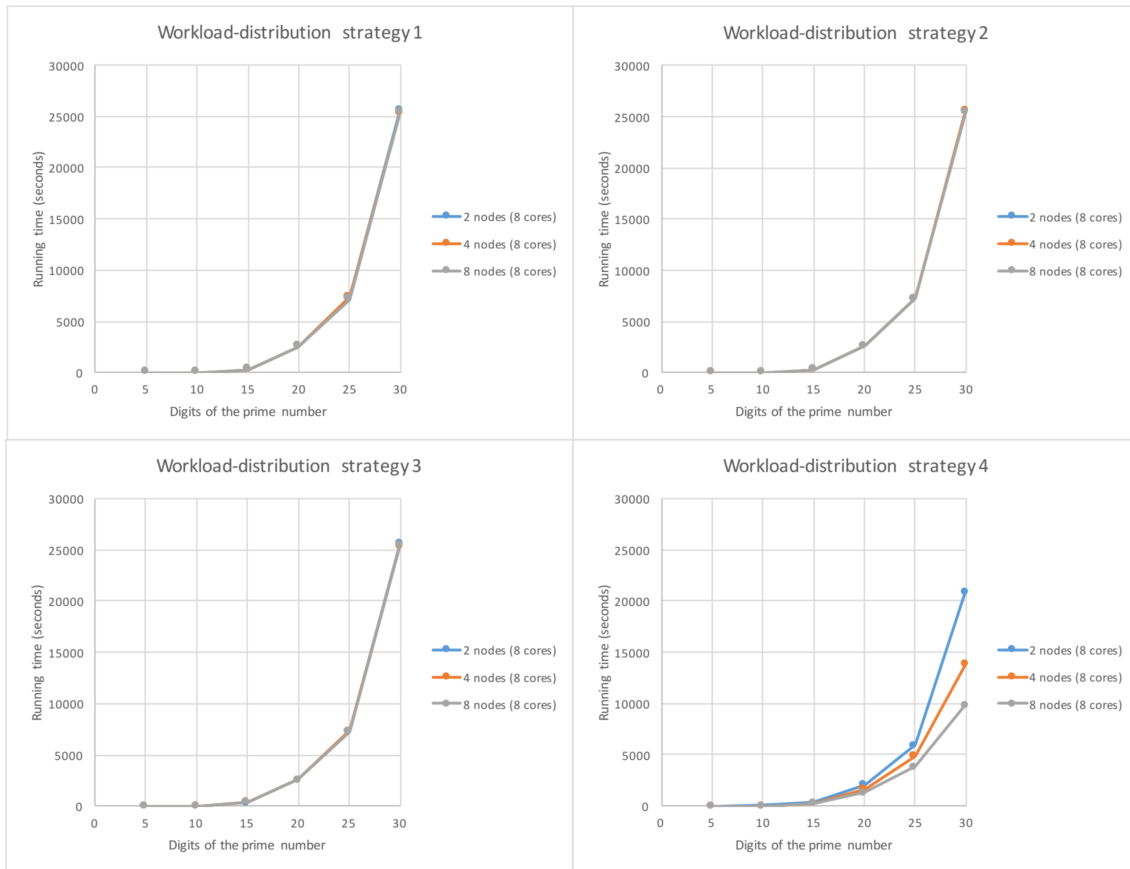

Figure 6. Running times for various numbers of cores per node

is for the effect to be so strong (1 core per node is faster more than twice compared to 4 cores per node). This means that an HPC with single-core nodes will produces even better results.

5. Conclusion

In this paper, we proposed a parallelization technique based on message passing parallelism for AKS algorithm. We also developed four workload-distribution strategies for this parallelization technique. From the experiments we have conducted, we conclude that dynamic workload-distribution strategy is the most load-balanced one. Furthermore, the difference between the dynamic strategy and static strategies is so significant that it is difficult to envision circumstances when one wishes to use the static ones. Overall, the dynamic strategy can achieve a speedup of up to 36 times the sequential computation. Nevertheless, the dynamic strategy has one obvious drawback, namely the bottleneck in the communication line towards the master node. The more nodes involved in the parallelism, the busier the master node and the heavier the communication line towards the master node. We did not manage to demonstrate this due to the limited size of the HPC available to us. We also showed that the number of cores per node has a strong effect for the dynamic workload-distribution strategy.

Acknowledgement

The authors would like to thank Directorate of Information System and Resources (DSSDI) of Universitas Gadjah Mada for providing the high-performance computing service used in this research.

References

[1] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978.
[2] H. Yang, "Research on design method based on hardware encryption and two-way id authentication for security mobile hard disk," *TELKOMNIKA*, vol. 12, no. 4, pp. 3001–3009, 2014.
[3] Z. Qi, "Secure digital certificate design based on the public key cryptography algorithm," *TELKOMNIKA*, vol. 11, no. 12, pp. 7366–7372, 2013.
[4] A. Muzakir and A. Ashari, "Rancang bangun keamanan web service dengan metode ws-security," *IJCCS (Indonesian Journal of Computing and Cybernetics Systems)*, vol. 6, no. 1, pp. 1–10, 2012.
[5] EFF, "EFF offers cooperative computing prizes," 2009, last accessed: 2017-03-19. [Online]. Available: https://www.eff.org/awards/coop
[6] G. L. Miller, "Riemann's hypothesis and tests for primality," *Journal of Computer and System Sciences*, vol. 13, no. 3, pp. 300–317, 1976.
[7] M. O. Rabin, "Probabilistic algorithm for testing primality," *Journal of Number Theory*, vol. 12, no. 1, pp. 128–138, 1980.
[8] C. Dong, "Math in network security: A crash course," 2016, last accessed: 2017-03-19. [Online]. Available: http://www.doc.ic.ac.uk/∼mrh/330tutor/
[9] R. Solovay and V. Strassen, "A fast Monte-Carlo test for primality," *SIAM Journal on Computing*, vol. 6, no. 1, pp. 84–85, 1977.
[10] M. Agrawal, N. Kayal, and N. Saxena, "PRIMES is in P," *Annals of Mathematics*, vol. 2, pp. 781–793, 2002.
[11] L. K. Nemana and V. C. Venkaiah, "An empirical study towards refining the AKS primality testing algorithm," *IACR Cryptology ePrint Archive*, vol. 2016, pp. 362–387, 2016.
[12] A. Granville, "It is easy to determine whether a given integer is prime," *Bulletin of the American Mathematical Society*, vol. 42, no. 1, pp. 3–38, 2005.
[13] H. W. Lenstra, "Primality testing with cyclotomic rings," Mathematic Institute, University of Leiden, Tech. Rep., 2002.
[14] D. Bernstein, "Proving primality after Agrawal-Kayal-Saxena," Department of Mathematics, Statistics, and Computer Science, University of Illinois, Tech. Rep., 2003.
[15] H. W. Lenstra, "Primality testing with Gaussian periods," in *FST TCS 2002: Foundations of Software Technology and Theoretical Computer Science, 22nd Conference Kanpur, India, December 12-14, 2002, Proceedings*, ser. Lecture Notes in Computer Science, M. Agrawal and A. Seth, Eds., vol. 2556. Springer, 2002, pp. 1–45.
[16] H. W. Lenstra and C. Pomerance, "Primality testing with Gaussian periods," Department of Mathematics, University of Dartmouth, Tech. Rep., 2011.
[17] D. Bernstein, "Proving primality in essentially quartic random time," *Mathematics of computation*, vol. 76, no. 257, pp. 389–403, 2007.
[18] R. E. Crandall and J. S. Papadopoulos, "On the implementation of AKS-class primality tests," University of Maryland College Park, Tech. Rep., 2003.
[19] H. Li, "The analysis and implementation of the AKS algorithm and its improvement algorithms," Master's thesis, Department of Computer Science, University of Bath, 2007.
[20] V. Menon, "Deterministic primality testing - understanding the AKS algorithm," *CoRR*, vol. abs/1311.3785, 2013.
[21] Z. Cao, "A note on the storage requirement for AKS primality testing algorithm," *IACR Cryptology ePrint Archive*, vol. 2013, pp. 449–452, 2013.
[22] C. K. Caldwell, "The prime pages: prime number research, records, and resources," 2017, last accessed: 2017-03-19. [Online]. Available: https://primes.utm.edu