A Multicore Exact Algorithm for Addition Sequence

Hazem M Bahig\textsuperscript{1,3*}, Yasser Kotb\textsuperscript{2,3}
\textsuperscript{1} College of Computer Science and Engineering, Hail University, Hail, KSA.
\textsuperscript{2} College of Computer and Information Sciences, Information Systems Department, Al Imam Mohammad ibn Saud Islamic University, Riyadh, KSA.
\textsuperscript{3} Computer Science Division, Department of Mathematics, Faculty of Science, Ain Shams University, Cairo, Egypt.

* Corresponding author. Email: h.bahig@uoh.edu.sa, ykkotb@imamu.edu.sa

Manuscript submitted August 22, 2018; accepted September 21, 2018.
doi: 10.17706/jcp.14.1.79-87

Abstract: This paper addresses one of the NP-complete problems that is used to reduce the number of computation for multi-exponentiation operation. The problem is the addition sequence for a set of n-positive integers $X = \{m_1, m_2, \ldots, m_n\}$ such that $2 \leq m_i < m_{i+1}$, $1 \leq i < n$ and $n \geq 2$. We suggest a parallel hybrid algorithm that finds a minimal length addition sequence for $X$. The algorithm traverses the search tree using breadth-first and branch-and-bound strategies. We also measure the effectiveness of the suggested algorithm on a multicore system and by using C++ language with OpenMP. The experimental results on sets of different sizes and different number of cores show that the proposed parallel hybrid algorithm for minimal length addition sequences is faster than the fastest sequential algorithm with maximum speed up 2.7.

Key words: Addition sequence, parallel algorithm, multicore, NP-complete, branch and bound, breadth search.

1. Introduction

Given a set of n-positive integers $X = \{m_1, m_2, \ldots, m_n\}$, such that $2 \leq m_i < m_{i+1}$, $1 \leq i < n$ and $n \geq 2$. The addition sequence, $AS$, for $X$ is a finite sequence of positive numbers such that [1], [2]:

\begin{align*}
as_0 &= 1, \\
as_i &= m_n, \\
as_i &= as_j + as_k, 1 \leq i \leq l \text{ and } 0 \leq j < k < i, \\
X &\subseteq \{as_0, as_1, as_2, \ldots, as_l\}.
\end{align*}

In case of $n=1$, $X = \{m_1\}$, the problem is called addition chain (AC). For fixed set $X$, many ASs can be generated with different lengths, $l$. For example, Table 1 lists different ASs for $X = \{5, 8, 19\}$. The first two sequences of length 7; while the last sequence of length 6.

If the length of $AS$ is minimal, then the sequence is called minimal length AS and the algorithm that finds a minimal length AS for $X$ is called exact algorithm. Otherwise, the length is not minimal and the sequence is called short $AS$. In both cases, generating short/minimal length AS is used to reduce/minimize the process of computing $\prod_{i=1}^{n} y^{m_i}$. Different cryptosystems used this strategy to compute multi-exponentiation,
where the base is $y [3]-[9]$. In this paper, we are interested to find an addition sequence with a minimal length, because the main drawback of this type is the execution time for generating a minimal length AS for $X$ is non-polynomial.

| Table 1. ASs for $X = \{5, 8, 19\}$ |
|-------------------------------------|
| AS for $X = \{5, 8, 19\}$          | Length |
|-------------------------------------|--------|
| $1, 2=1+1, 4=2+2, 5=4+1, 8=4+4, , 16=8+8, 18=16+2, 19=18+1$ | 7      |
| $1, 2=1+1, 4=2+2, 5=4+1, 8=4+4, , 9=8+1, 11=9+2, 19=11+8$ | 7      |
| $1, 2=1+1, 3=2+1, 5=3+2, 8=4+4, , 11=8+3, 19=11+8$ | 6      |

Over many years ago, computer architectures have developed significantly and these developments are ubiquitous today. One of these developments is building a high-performance system. In such system, we use many CPUs or processors/cores concurrently to run many threads. The main goal of this system is increasing the performance of applications and solving the scientific and engineering problems in a fast time. Examples of this system are graphics processing unit, multi-core, clusters, grid, and cloud.

The goal of our work is to design a parallel algorithm on the multi-core system to speed up the execution time of generating a minimal length AS. Another reason for this work, there is no previous parallel algorithms for finding a minimal length addition sequence until now. The designed algorithm is based on hybrid strategy: breadth and branch-and-bound strategies. The efficiency of the designed parallel algorithm was measured on a multicore system consisting of 12 cores using C++ language with OpenMP.

The work of this paper consists of an introduction and four sections. In the second section, we introduce briefly the literature review for AS and the outlines of the best known algorithm for minimal length AS. Section 3 discusses the main steps of the designed parallel AS algorithm. Section 4 contains the discussion and the experimental results of the designed algorithm. Finally, we conclude our work in Section 5.

2. Related Work

Several sequential algorithms for AS have been introduced to generate short AS [10]-[12] and minimal length AS [1], [2], [4], [13], [14]. The proposed sequential algorithms for short AS are based on continued fraction and binary methods. The running time for these algorithms is fast, but the length of the sequence is not minimal. On the other side, for a minimal length AS, the authors in [2] proved that the AS problem is NP-complete. For some special sets, the authors in [1] proposed a method to find minimal length AS. Also with some limitations and restrictions, Bleichenbacher [3] suggested a special method to find for a minimal length AS for $X$. Both methods are not suitable for the general case of AS problem.

Bahig [13], proposed exact algorithm to generate a minimal length AS based on branch-and-bound method. The algorithm was refined by the same authors in [14] to find the exact solution in a faster time. The improvements include bounding sequences, and conditions and lower bounds for some types of steps. As far as we know the best proposed algorithm for minimal length AS is the algorithm mentioned in [14]. On other side, no parallel algorithms are suggested to generate AS with minimal length until now except for a short addition chain [15]. Therefore, our target in this work is to study the minimal length AS using the parallelism. The reasons for this direction of research are no parallel algorithm for a minimal length AS and to reduce the time consumed for generating a shortest sequence.

The outlines of the branch-and-bound algorithm [14] for minimal length AS are shown in Algorithm 1. The algorithm is based on many subroutines that are used to calculate lower and upper bounds, and bounding sequences. These subroutines are:
1. Lower bounds subroutine LB as in [14]. The input of this subroutine is a set of positive integers \( X = \{m_1, m_2, \ldots, m_n\} \), such that \( 2 \leq m_i < m_{i+1} \), \( 1 \leq i < n \).

2. Lower bounds subroutine LB as in [14]. The input of this subroutine is a set of positive integers \( X = \{m_1, m_2, \ldots, m_i\} \), where \( Lb = Lb_n \) is the lower bound for \( l(X) \).

3. Upper bound subroutine UB as in [14]. The input of this subroutine is a set of positive integers \( X = \{m_1, m_2, \ldots, m_i\} \). The output of the subroutine is an upper bound \( Ub \) using the continued fraction method.

4. Bounding sequences subroutine BS as in [14]. The input of this subroutine is a set of positive integers \( X = \{m_1, m_2, \ldots, m_i\} \) and a sequence of lower bounds \( Lb_i \) of \( l(X_i) \), where \( 1 \leq i \leq n \). The output of the subroutine is a set of bounding sequences \( bs_i \) for \( 0 \leq i \leq Lb_n \).

3. The Parallelization Method

This section aims to present a parallelization strategy to find a minimal length AS. The parallelization method for a minimal length AS uses a hybrid strategy of two search methods: breadth-first search and B2AS algorithm. In the breadth search method, we visit the neighbor nodes at the current level first, before traversing to the next level neighbors.
The proposed parallel algorithm, Parallel Breadth-Branch-Bound AS (PB3AS), begins the process of generating the sequence by determining the lower and upper bounds for the goal \( X \) and assigns initially the current length, \( l \), of the sequence with the value of the lower bound for \( X \). This process represents Step 1 in Algorithm 2, while Step 2 represents the calculation of the bounding sequences for \( X \) that is used to reduce the size of the solution space by deleting some elements of the search space. After that we execute the strategy of breadth-first search from level 1 to level \( \beta \). The value of \( \beta \) is small because the breadth search will generate large number of nodes when the level is large and therefore, the algorithm will required large amount of memory. The initial sequence at level 1 is \( \{2, 1\} \). The generated elements at level 2 are 3 and 4, so the two partial sequences from root to the level 2 are \( \{3, 2, 1\} \) and \( \{4, 2, 1\} \). The generated elements at level 3 are 4, 5, and 6 from the path \( \{3, 2, 1\} \), and 5, 6, and 8 from the path \( \{4, 2, 1\} \). Therefore, all partial sequences from the root to level 3 are \( \{4, 3, 2, 1\} \), \( \{5, 3, 2, 1\} \), \( \{6, 3, 2, 1\} \), \( \{5, 4, 2, 1\} \), \( \{6, 4, 2, 1\} \), and \( \{8, 4, 2, 1\} \).

Similarly, we can generate all elements until level \( \beta \). Each element at level \( \beta \) represents as a sequence of numbers from root \( as_0 \) to the level \( \beta \), \( \{as_0, as_{\beta-1}, ..., as_2, as_1, as_0\} \). The process of generating all paths from the root to level \( \beta \) represents Step 3 in Algorithm 2.

After generating all possible partial ASs from level 0 to \( \beta \) by a sequential manner, we complete the searching process using the B2AS algorithm in a parallel manner. The parallelization of this stage can be done as follows. Assume that the total number of threads is \( n_t \) and the total number of elements at level \( \beta \) is \( n_\beta \) with assuming that \( n_t \ll n_\beta \). Initially, the algorithm will assign the first \( n_t \) elements at level \( \beta \) to the \( n_t \) threads as in Step 4.1 for Algorithm 2. Each thread executes B2AS algorithm on the assigned element and

---

**Algorithm 2: PB3AS (Parallel Breadth-Branch-Bound AS)**

**Input:** A set of \( n \) positive integer numbers \( X = \{m_1, m_2, \ldots, m_n\} \), such that \( 2 \leq m_i < m_{i+1} \), \( 1 \leq i < n \).

**Output:** Minimal length AS, \( a_{s_1}, a_{s_2}, \ldots, a_{s_l} \).

**Begin**

1. **Lower and upper bounds:** Calculate for the set of integer \( X \) the lower bound, \( Lb \), and the upper bound, \( Ub \) using subroutines LB and UB as in [14].

2. **Bounding sequences:** Calculate the vertical and slant bounding sequences by using BS subroutine as in [14].

3. **Breadth-first Search:** Generate a breadth tree, sequentially, starting from the level 1, that represents the sequence \( as_0=1, as_1=2 \), to the level \( \beta \). The generated elements are stored in dynamic array, \( D \), each element of the dynamic array consisting of the path of the partial sequence from level 0 to level \( \beta \).

4. **B&B Search:** Repeat the following dynamically in the parallel until no elements in the dynamic array or the goal is found
   4.1 Assign the available thread to one element from the dynamic array, say \( a_q \).
   4.2 Apply the B2AS algorithm to find the goal \( X = \{m_1, m_2, \ldots, m_n\} \) starting from \( a_q \). If a thread finds the sub-goal \( m_i \) (initially \( i=1 \)) within the determined lower bound, where \( m_i \) is the current sub-goal. Then we test if the current sub-goal is not the final sub-goal, \( m_n \), then we increment \( i \) by one and continue the process, B&B-AS algorithm, to find the new sub-goal, \( m_{i+1} \). Otherwise (final sub-goal, \( m_n \) is found), terminate the process of searching and return with the shortest addition sequence. Otherwise (the thread fails to find the current goal with the lower bound), return to step 4.1 and assign a new element from the dynamic array, if not empty, to the thread.

5. **Updating bound:** Adding one to the lower bound, sequentially, and return back to execute steps 2-4.

**End**
will try to search the elements of the goal \( X, m_1, m_2, \ldots, m_n \) one by one such that the length of the sequence is \( L_b \) as in Step 4.2 for Algorithm 2. For more details, each thread will try to find the first subgoal \( m_1 \) within the calculated lower bound \( L_b \). If the result of searching is true, then the thread will try to find the second subgoal \( m_2 \) and so on. In general, each thread will try to find the sub-goal \( m_j \), \( 1 \leq j < n \), within the calculated lower bound, and if the result of search is true, then the thread will try to find a new subgoal \( m_{j+1} \). If the last element of goal \( X, m_n \) exist with length \( L_b \), then the thread terminates the process of searching and outputs with a minimal length AS. If the result of the B2AS algorithm is not found, this means that the thread cannot find a sequence of elements contains the elements of \( X \) with length \( L_b \). In this case, the algorithm will assign another element, if exist, from the elements of level \( \beta \) to the thread and then do the same process. If all elements generated by breadth search step cannot lead to the final goal, then the length of shortest sequence should be changed to \( L_b+1 \). In any iteration of Step 4, if one of the threads finds the set \( X \) in a sequence of elements with length \( L_b \), then the thread will return with the minimal length sequence and the process of searching terminated and all other threads will terminated.

The main part of PB3AS algorithm that effects on the running time is Step 4. The execution time for all others steps can be neglected compared to the time of Step 4. This part affected by how many elements generated at level \( \beta \), the size of the set \( X \), the number of threads used concurrently, and the difference between the initial value of the lower bound and the length of the minimal length AS.

4. Experiment Results and Analysis

This section describes the results of the experiments for the presented work and then we discuss and analysis these results. The section consists of three parts. In the first part, we briefly state the configuration of the platform and data generation. The second part contains the measurements of the running time and storage for the presented algorithm, while the speedup and scalability measurements are discussed in the third part.

4.1. Platform Configuration and Data Generation

We implemented the proposed AS parallel algorithm (PB3AS) and the best known sequential algorithm (B2AS) using the following specification of platforms.

1. The computer used in the experiments consists of 12 cores of speed 2.6 GHz and a memory of size 16 G Bytes. The computer is worked under operating system Window 8.
2. The language used to convert the algorithm to a code is the C++ language and the parallel regions of the program were written using OpenMP directions and library routines.

We generated the elements of the set \( X \) from the interval \([2^8, 2^{16}]\). The sizes of the set \( X \) were tested are 2, 3, and 4. For fixed size of \( X \), we generate 50 random numbers. Also, we used only \( p=1, 4, 8 \) and \( 12 \) cores when the size of \( X \) is 2 and 3. On the other side, when the size of \( X \) is 4, we ran the proposed parallel algorithm when \( p=12 \). Because the running time of the sequential program for AS is large. Also, we use two values for the number of levels for breadth search \( \beta=5 \) and 8. If we increase the value of \( \beta \) more than 8, then the required storage is very large in some instances.

4.2. Execution Time and Storage

In this part, we measured the execution time and storage of the designed PB3AS algorithm experimentally. Moreover, it compared with B2AS algorithm using the platform configuration and dataset stated in the previous part. No comparison with previous parallel AS algorithm, because until writing this work, no previous parallel AS algorithms. The results of these experiments are shown in the Figs. 1, 2 and 3 which show the running times for B2AS and PB3AS algorithms when the sizes of the set \( X \) are 2, 3 and 4. For fixed value of \( p \geq 4 \), we have two bars at
$p$ in each figure to represent the running times of the PB3AS algorithm when $\beta=5$ and 8 for the same size of set $X$. Otherwise, $p=1$, we have only one bar in the figures.

The analyses of these data results that are shown in Figs. 1-3 indicate the following observations related to the running time.

1. The execution time of PB3AS algorithm is faster than B2AS algorithm for all data sets. For example, when the size of $X$ is 3, the running time for B2AS algorithm (sequential) is 11281 seconds, while the running times for PB3AS algorithm are 7834, 6033, and 5397 seconds using 4, 8, and 12 cores, respectively in case of $\beta=5$.

2. If the size of $X$ and the number of breadth level $\beta$ are fixed, and the number of cores is varied, then the running time of the PB3AS algorithm decreases with increasing the number of cores. For example, if the size of $X$ is 2 and the value of $\beta$ is 8, then the execution times of PB3AS algorithm are 549.8, 502.4, and 454 using 4, 8 and 12 cores, respectively.
3. If the size of $X$ and the number of cores are fixed, and the number of level for breadth search is varied, then the running time of the PB3AS algorithm decreases with increasing the number of levels. For example, if the size of $X$ is 4 and the number of cores is 12 then the execution times of the PB3AS algorithm are 24033.7 and 18762.4 seconds for the breadth level $\beta=5$ and 8, respectively.

4. If the number of cores and the breadth level are fixed, and the size of $X$ is varied, then the running time of the PB3AS algorithm increases with increasing the size of $X$. For example, if the number of cores is 12 and the number of breadth level is 8 then the execution times of the PB3AS algorithm are 454, 4424.2, and 18762.4 seconds for the sizes of the set $X$ equal 2, 3 and 4, respectively.

5. The execution time for breadth step is very small compared to branch-and-bound strategy, and the amount of this time is less than 0.1% of the total time in the most cases.

Additionally, the experiments demonstrate the following observations related to storage.

1. The storage required by PB3AS algorithm is larger than B2AS algorithm for two reasons. The first is the PB3AS algorithm uses the breadth strategy. The second is there are $n_t$ (number of threads) different paths worked concurrently in PB3AS algorithm and each thread has stack size to store the elements of the searching.

2. In PB3AS algorithm, the storage required for breadth step in case of $\beta=5$ is less than $\beta=8$. On the other side, the storage required for B&B search (Step 4) in the case of $\beta=5$ is greater than $\beta=8$. In general, the total amount of storage required for PB3AS algorithm in the case of $\beta=5$ is less than $\beta=8$.

3. For increasing the size of $X$ and the number of 1's for the binary representation of each sub-goal, the storage required by PB3AS algorithm increased in general.

4. Increasing the number of levels greater than 8 leads to increase the storage of PB3AS algorithm rapidly.

### 4.3 Speedup Measurement

The principal measure for the effectiveness of parallelization is the speedup, $S_p$, which is computed by using the following equation, where $T_s$ and $T_p$ are the execution time for sequential and parallel algorithm respectively.
Table 2. Speedup of the PB3AS Algorithm

| $n$ | $p$ | 5  | 8  | 12 | 5  | 8  | 12 |
|-----|-----|----|----|----|----|----|----|
| 2   |     | 1.25 | 1.59 | 1.83 | 1.8 | 1.97 | 2.18 |
| 3   |     | 1.44 | 1.87 | 2.09 | 2.01 | 2.41 | 2.55 |
| 4   |     | --   | --   | 2.1  | --  | --   | 2.69 |

Fig. 4. Scalability of the PB3AS algorithm.

\[
S_p = \frac{T_1}{T_p}
\]

Table 2 shows that the speedup of the PB3AS algorithm increases with the following cases.

1. Increasing the number of cores. For example, if the size of $X$ is 3, the speedup of PB3AS algorithm using $p=4$, 8, and 12 are 1.4, 1.8, and 2, respectively.

2. Increasing the size of $X$. For example, the speedup of the PB3AS algorithm when the sizes of $X$ are 2, 3, and 4 using 12 cores and number of level equals 5 are 1.8, 2.1 and 2.7, respectively.

The maximum value of speedup achieved by the proposed algorithm is around 2.7 times the best sequential B2AS algorithm. Based on the value of speedup, we can compute the scalability of the PB3AS algorithm as shown in Fig. 4.

5. Conclusion

In this work, we used the parallel computing concepts to study one of the NP-complete problems. This problem addresses the generation of a minimal length addition sequence for a set of positive numbers $X$. We have presented a new parallel algorithm for AS. The presented algorithm uses hybrid strategy to generate a minimal length AS. The first strategy uses the breadth search as sequential step to generate many elements in the search space. Then we used the B&B search in parallel to find a shortest sequence. Also, we have studied the presented algorithm experimentally on a computer consisting of 12 cores and implemented it using C++ language and OpenMP. The results of the experiments indicate that the execution time of the presented parallel algorithm is faster than the best known sequential algorithm. The maximum speedup achieved by the presented algorithm is 2.7 times the best known sequential algorithm.

Acknowledgment

This paper contains the results and funding of a research project that is funded by King Abdulaziz City for Science and Technology (KACST), Grant No. M S – 36 - 185.
References

[1] Chen, Y., Chang, C., & Yang, W. (1994). Some properties of vectorial addition chains. *Int J Comput Math*, 54, 185-196.

[2] Downey, P., Leony, B., & Sethi, R. (1981). Computing sequences with addition chains. *SIAM Journal of Computing*, 3, 638-696.

[3] Bleichenbacher, D. (1996). *Efficiency and Security of Cryptosystems Based on Number Theory*. Unpublished doctor thesis, Swiss Federal Institute of Technology Zurich, Zurich.

[4] Knuth, D. (1973). *The Art of Computer Programming: Seminumerical Algorithms*, 2. Addison-Wesley.

[5] Laih, C., Yen, S., & Harn, L. (1991). Two efficient server-aided secret computation protocols based on the addition sequence. *Advances in Cryptology — ASIACRYPT ‘91, LNCS*, 739, 450-459.

[6] Schönhage, A. (1975). A lower bound for the length of addition chains. *Theor Comput Sci*, 1, 1-12.

[7] Chang, C., & Lou, D. (1997). Parallel computation of the multi-exponentiation for cryptosystems. *International Journal of Computer Mathematics*, 63, 9-26.

[8] Laih, C., & Yen, S. (1992). Secure addition sequence and its applications on the server-aided secret computation protocols. *Advances in Cryptology — AUSCRYPT ‘92, LNCS*, 718, 217-230.

[9] Yen, S., & Laih, C. (1992). The fast cascade exponentiation algorithm and its applications on cryptography. *Advances in Cryptology — AUSCRYPT ’92, LNCS*, 718, 447-456.

[10] Bergeron, F., Berstel, J., Brlek, S., & Duboc, C. (1989). Addition chains using continued fractions. *Journal of Algorithms*, 10, 403-412.

[11] Bergeron, F., Berstel, J., & Brlek, S. (1994). Efficient computation of addition chains. *J Theor Nombres Bord*, 6, 21-38.

[12] Rooij, P. (1995). Efficient exponentiation using precomputation and vector addition chains. *Advances in Cryptology – EUROCRYPT’94, Lecture Notes in Computer Science*, 950, 389-399.

[13] Bahig, H., & Bahig, H. (2006). Speeding up evaluation of powers and monomials. *Proceedings of the 2006 International Conference on Foundations of Computer Science* (pp. 149-153). Las Vegas, Nevada, USA, June 26-29.

[14] Bahig, H., & Bahig, H. (2011). A new strategy for generating shortest addition sequences. *Computing*, 91(3), 285-306.

[15] Bahig, H. (2018). A fast optimal parallel algorithm for a short addition chain. *J Supercomputing*, 74(1), 324-333.

Hazem M Bahig is an associate professor in College of Computer Science and Engineering, Hail University, KSA. He received MSc and PhD degrees in computer science in 1997 and 2003 respectively from Ain Sham University, Egypt. His current research interests include high performance computing, algorithm, bioinformatics, and e-learning systems for algorithms.

Yasser Kotb is an assistant professor with College of Computer and Information Sciences, Al Imam Mohammad Ibn Saud Islamic University (IMSIU), KSA (On leave from Faculty of Science, Ain Shams University, Egypt). He received the PhD degree from School of Information Science, Japan Advanced Institute of Science and Technology (JAIST), Japan (2003). His research interests include bioinformatics, data and web mining, web technologies, parallel algorithm, and formal methods.